`ALIAS-C++`

A C++ Algorithms Library of Interval Analysis for equation Systems

Version 2.5

September 2007

The COPRIN project

Download ALIAS here!

Chapter 1

Introduction

`ALIAS`-C++ is a C++ library of algorithms, intended to be run on Sun computers and PC under Unix or Linux, that deal with problems like solving systems of equations and inequalities,optimization, linear algebra .... Expressions involved in these problems may be arbitrary combination of the most classical mathematical functions (algebraic terms, sine, cosine, log etc..) and whose coefficients are real numbers or, in some cases, intervals. Most algorithms in `ALIAS` are based on interval analysis and can be used for almost any system as soon as it is composed of classical mathematical operators (see section 2.1.1.2 for the available operators). Some algorithms may be used only for problems with specific structure such as solving algebraic, linear, distance, ... equation systems. `ALIAS` may also deal with functions that involve determinants of matrices, without having to expand the determinants. Without being exhaustive you may find in `ALIAS` algorithms that enable one to:

- find an approximation of the real roots of 0-dimensional systems
- find an approximation of the variety defined by n-dimensional systems
- find an approximation of the *global* minimum or maximum of a function (eventually under equations and/or inequalities constraints) up to an accuracy provided by the user
- analyze a system of algebraic equations to determine bounds for its real roots
- parse a formula file describing a set of functions in order to compute an interval evaluation of the functions that can be used for the solving procedures
- check the regularity of an interval matrix
- provide an enclosure of definite integrals

This version of the library has about 86353 lines of C++ code, to which should be added the 13 000 lines for `BIAS/Profil`/ Most of these algorithms are designed in view of their use in a parallel implementation (see chapter 13).

This library is one of the main software development platform of the COPRIN project. There are various motivations underlying the development of this platform:

- interval analysis allows to solve very different problems (not only solving systems of equations) as soon as the unknowns are restricted to lie within a given domain, which is relevant to many applications. The expertise of the COPRIN project in application domains such as mechanism theory, control theory, robotics has allowed us to develop algorithms and test the library on numerous realistic problems (historically speaking it was the need of tools to solve problems in these fields that has led to the development of `ALIAS`)
- interval analysis allows to deal with uncertainties that are present in almost all applications
- interval analysis allows to take into account numerical round-off errors that may have a very negative impact on the result of classical numerical analysis algorithms. This allows one to get *safe* results: whenever an interval-analysis based method will give a solution then the result is guaranteed. On the

other hand such algorithm may just states that the current problem cannot be solved with the current computer arithmetics. At the same time interval-analysis based methods may provide an *exact* answer in the sense that it will be possible to get a solution of the problem wit an arbitrary accuracy in term of number of exact digits for the solution.

- classical problem solving approaches rely on the transformation of the initial problem into a solving problem that have the same solution (or solutions that may be transformed into the solutions of the initial problem) but which has a structure that is more appropriate for solving (for example a system of sine and cosine will be transformed into an algebraic system for which there is powerful solving methods). This operation may transform a simple problem into a much more complex solving problem. At the opposite interval analysis allows one to focus on the problem to be solved which will often results in a more efficient strategy This may also be considered a drawback of interval analysis: very often it is necessary to think backward and to focus on what is really the problem to be solved in view of a solving by interval analysis. We will present some examples of this approach.

- although in many cases other approaches may be used to solve problems addressed in `ALIAS` they are very often computer intensive. A short pre-processing with interval analysis based methods may avoid using these approaches whenever it is not necessary, therefore drastically reducing the whole computation time. Furthermore the computation time of interval analysis based methods will decrease with the decrease of the search space: hence there will always be a limit on the search space under which the interval analysis method will be faster than any other method

- although they are numerous papers and books on interval analysis (see [5, 6, 17, 18, 20, 21, 24] to name a few) the few implementations that are freely available are far from being complete. Our purpose is to offer a wide range of algorithms

- various communities are working on interval analysis based algorithms: constraint programming, operational research, numerical analysis, ... Furthermore other communities (such as the algebraic geometry community) are addressing the same problems. In our view there is a lack of collaboration between these communities although basically they intend to solve the same type of problems and that the different methods are very often complementary

- the ALIAS-C++ library is partly interfaced with a Maple package. We strongly believe that symbolic computation, apart for being very convenient to produce error-free C++ code, will also play an important role to improve the efficiency of the algorithms. The ALIAS-Maple manual provides procedures and examples for this increase in efficiency

- we do not claim that interval analysis is an universal tool: it may fail to solve a given problem. It is almost impossible to determine beforehand if interval-analysis based method will work. One of the purposes of `ALIAS` is to provide the necessary tool to test quickly existing methods. Another purpose is to provide the basic tools so that new algorithms may be developed efficiently. On the other hand interval analysis allow to solve problems for which (to the best of our knowledge) there is no alternate solution.

- we have sometime heard the somewhat negative comment that interval analysis is "only" a version of the branch-and-bound algorithm. Although branch-and-bound is indeed a key component of the interval analysis algorithm we strongly believe that classifying the strategy only by this component is unfair as many other methods are used (and will make a drastic difference in term of computation time). Furthermore we consider that methods should also be judged according to what problem they may solve. In that regard interval analysis is very rich and has proved to be effective in realistic applications problems for which, to the best of our knowledge, there was no alternate methods (chapter 15 will present some of this problems)

`ALIAS` is partly interfaced with Maple, see the ALIAS-Maple documentation.
This manual contains the following chapters:

- Chapter 2 is devoted to a brief description of interval analysis and to the various algorithms of `ALIAS` enabling to solve systems, either general purpose solving methods or algorithms for specific type of systems (e.g. system with a large number of linear terms or involving determinant of matrices).

- Chapter 3 deals with the analysis of system of functions
- Chapter 4 deals especially with the analysis of trigonometric functions
- Chapter 5 describes algorithms for the analysis of univariate polynomial.
- Chapter 6 is devoted to the study of parametric polynomials and calculation on the eigenvalues of parametric matrices
- Chapter 7 is devoted to procedure related to linear algebra
- Chapter 8 deals with global optimization problems, constrained or not.
- In Chapter 9 we describe how `ALIAS` can be used for dealing with one-dimensional systems and $n > 0$ dimensional systems
- Chapter 10 is devoted to guaranteed integration procedures
- Chapter 11 describes some miscellaneous `ALIAS` procedures
- Chapter 12 is devoted to the `ALIAS` parser and the generic analyzer and solver based on this parser
- In Chapter 13 we explain how `ALIAS` can be used in a parallel implementation
- In Chapter 14 we explain how to use and install `ALIAS`,
- the examples treated in the various chapters are presented in chapter 12,
- Chapter 15 presents various examples used in this documentation together with some examples of problems that interval analysis can solve
- Chapter 16 is the troubleshooting chapter, in which is explained what can go wrong with `ALIAS` procedures and how to adjust `ALIAS` parameters to obtain a better efficiency

1.1 How to read this manual

This manual is intended to be read by two types of users:

- application oriented
- developer

In the first case you may skip all the "Mathematical background" sections and go directly to the "Implementation" and "Examples" sections. In some cases it may be necessary however to consult the "Mathematical background" section to understand the meaning of some parameters of the procedure you intend to use.

In the latter case the "Mathematical background" sections may be of interest.

Chapter 2

Solving with Interval Analysis

2.1 Introduction

The purpose of this chapter is to describe the methods based on interval analysis available in the `ALIAS` library for the determination of real roots of system of equations and inequalities.

2.1.1 Interval Analysis

2.1.1.1 Mathematical background

This section is freely inspired from the book [5]. An interval number is a real, closed interval $(\underline{x}, \overline{x})$. Arithmetic rules exist for interval numbers. For example let two interval numbers $X = (\underline{x}, \overline{x})$, $Y = (\underline{y}, \overline{y})$, then:

$$X + Y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$
$$X - Y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$$

An *interval function* is an interval-valued function of one or more interval arguments. An interval function F is said to be *inclusion monotonic* if $X_i \subset Y_i$ for i in $[1, n]$ implies:

$$F(X_1, \ldots, X_n) \subset F(Y_1, \ldots, Y_n)$$

A fundamental theorem is that any rational interval function evaluated with a fixed sequence of operations involving only addition, subtraction, multiplication and division is inclusion monotonic. This means in practice that the interval evaluation of a function gives bounds (very often overestimated) for the value of the function: for any specific values of the unknowns within their range the value of the function for these values will be included in the interval evaluation of the function. A very interesting point is that the above statement will be true even taking into account numerical errors. For example the number 1/3, which has no exact representation in a computer, will be represented by an interval (whose bounds are the highest floating point number less than 1/3 and the smallest the lowest floating point number greater than 1/3) in such way that the multiplication of this interval by 3 will include the value 1. A straightforward consequence is that if the interval evaluation of a function does not include 0, then there is no root of the function within the ranges for the unknowns.

In all the following sections an interval for the variable x will be denoted by $(\underline{x}, \overline{x})$. The *width* or *diameter* of an interval $(\underline{x}, \overline{x})$ is the positive difference $\overline{x} - \underline{x}$. The *mid-point* of an interval is defined as $(\overline{x} + \underline{x})/2$.

A *box* is a set of intervals. The width of a box is the largest width of the intervals in the set and the *center* of the box is the vector constituted with the mid-point of all the intervals in the set.

2.1.1.2 Implementation

All the procedures described in the following sections use the free interval analysis package `BIAS/Profil`[1] in which the basic operations of interval analysis are implemented[2].

[1] http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html

[2] Some bugs in this package have been corrected and the patches are available in the version distributed with the `ALIAS` package

This package uses a fixed precision arithmetics with an accuracy of roughly 10^{-16}. Different types of data structure are implemented in this package. For fixed value number:

`BOOL, BOOL_VECTOR, BOOL_MATRIX,`
`INT, INTEGER_VECTOR, INTEGER_MATRIX, REAL, VECTOR, MATRIX, COMPLEX`

for intervals:

`INTERVAL, INTERVAL_VECTOR, INTERVAL_MATRIX`

All basic arithmetic operations can be used on interval-valued data using the same notation than for fixed numbers. Not that for vector and matrices the index start at 1: $V(1,1)$ for an interval matrix represents the interval at the first row and first column of the interval matrix V. The type `INTERVAL_VECTOR` will be used to implement the box concept, while the type `INTERVAL_MATRIX` will be used to implement the concept of list of boxes.

For the evaluation of more complex interval-valued function there are also equivalent function in the BIAS/Profil, whose name is usually obtained from their equivalent in the C language by substituting their first letter by the equivalent upper-case letter: for example the evaluation of $\sin(X)$ where X is an interval will be obtained by calling the function $Sin(X)$. We have also introduced in `ALIAS` some other mathematical operators whose names are derived from their Maple implementation: `ceil, floor, round`.

Table 2.1 indicates the substitution for the most used functions.

C function	Substitution	C function	Substitution
sin	Sin	cos	Cos
tan	Tan	arcsin	ArcSin
arccos	ArcCos	arctan	ArcTan
sinh	Sinh	cosh	Cosh
tanh	Tanh	arcsinh	ArcSinh
arccosh	ArcCosh	arctanh	ArcTanh
exp	Exp	log	Log
log10	Log10	x^2	Sqr
sqrt	Sqrt(x)	$\sqrt[i]{(x)}$	Root(x,i)
x^i	Power(x,i)	x^y	Power(x,y)
$\vert x\vert$	IAbs(x)	ceil(x)	ALIAS_Ceil(x)
floor(x)	ALIAS_Floor(x)	rint(x)	ALIAS_Round(x)

Table 2.1: Equivalent interval-valued function

Note also that the mathematical operators *cot*, *arccot*, *arccoth* exist under the name `Cot`, `ArcCot`, `ArCoth`.

A special operator is defined in the procedure `ALIAS_Signum`: formally it defines the `signum` operator of Maple defined as

$$\mathrm{signum}(x) = \frac{x}{|x|}$$

which is not defined at $x = 0$. In our implementation for an interval $x = [\underline{x}, \overline{x}]$ `ALIAS_Signum`(x) will return:

- 1 if $\underline{x} > 0$
- -1 if $\overline{x} < 0$
- 1 if $|x|$ is lower than `ALIAS_Value_Sign_Signum` and `ALIAS_Sign_Signum` is positive
- -1 if $|x|$ is lower than `ALIAS_Value_Sign_Signum` and `ALIAS_Sign_Signum` is negative

-1,1 otherwise

The default value for `ALIAS_Value_Sign_Signum` and `ALIAS_Sign_Signum` are respectively 1e-6 and 0.

The derivative of `ALIAS_Signum` is defined in the procedure `ALIAS_Diff_Signum`. Formally this derivative is 0 for any x not equal to 0. In our implementation `ALIAS_Diff_Signum`(x) will return 0 except if $|x|$ is lower than `ALIAS_Value_Sign_Signum` in which case the procedure returns [-1e11,1e11].

The derivative of the absolute value is defined in the procedure `ALIAS_Diff_Abs`. If the interval `X` includes 0 the procedure returns [-1e11,1e11] otherwise it returns `ALIAS_Signum(X)`.

Using the above procedures when an user has to write an interval-valued function he has to convert its C source code using the defined substitution. For example if a function is written in C as:

$$double\ x, y, z, f; \quad f = sin(x) * cos(y) + sqrt(z);$$

then its equivalent interval valued function is

$$INTERVAL\ x, y, z, f; \quad f = Sin(x) * Cos(y) + Sqrt(z);$$

Note that all the even powers of an interval are better managed with the `Sqr` and `Power` procedures. Indeed let consider the interval $X = [-1, 1]$, then the interval product $X \times X$ leads to the interval [-1,1] while the interval `Sqr`(X) leads to [0,1]. For an interval $X = (\underline{x}, \overline{x})$ the width of the interval is obtained by using the procedure `Diam`(X) while we have $\underline{x}$ =`Inf`(X) and $\overline{x}$ =`Sup`(X).

We will denote by *box* a set of intervals which define the possible values of the unknowns. By extension and according to the context *boxes* may also be used to denote a set of such set. A *function intervals* will denote the interval values of a set of functions for a given box, while a *solution intervals* will denote the box which are considered to be solution of a system of functions.

2.1.1.3 Problems with the interval-valuation of an expression

An important point is that not all expressions can be evaluated using interval arithmetics. Namely constraints that prohibits the interval evaluation of an expression are:

- 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 include the interval [-1,1]
- argument x of x^y where y is not an integer should be
- argument x of $exp(x)$ should not be too large to avoid overflow problem.

If such situation occurs a fatal error will be generated at run time. Hence such special cases has to be dealt with carefully. ALIAS-Maple offers the possibility of dealing with such problem. For example the procedure `Problem_Expression` allows one to determine what constraints should be satisfied by the unknowns so that each equations can be interval evaluated, see the ALIAS-Maple documentation.

If you use your own evaluation procedure and are aware of evaluation problems and modify the returned values if such case occurs it will be a good policy to set C++ flags `ALIAS_ChangeF`, `ALIAS_ChangeJ` to 1 (default value 0) if a change occurs. Currently the interval Newton scheme that is embedded in some of the solving procedures of `ALIAS` will not be used if one of these flags is set to 1 during the calculation.

2.1.1.4 Dealing with infinity

In some specific cases we may have to deal with interval in which infinity is used. These quantities are represented using BIAS convention, `BiasNegInf` representing the negative infinity and `BiasPosInf` the positive infinity.

2.2 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. In that case the result will be a set of boxes which will be an approximation of the solution. When dealing with such system it is necessary to set the global variable `ALIAS_ND` to 1 (its default value is 0) and to define a name in the character string `ALIAS_ND_File`. The solution boxes of the system will be stored in a file with the given name. The quality of the approximation may be estimated with the flags `ALIAS_Volume_In`, `ALIAS_Volume_Neglected` that give respectively the total volume of the solution boxes and the total volume of the neglected boxes (i.e. the boxes for which the algorithm has not been able to determine if they are or not a solution of the system). Note that there are special procedures for 1-dimensional system, see chapter 9.

2.3 General purpose solving algorithm

This algorithm enable to determine approximately the solutions of a system of $n > 0$ equations and inequalities in $m > 0$ unknowns. Hence this method may be used to solve a system composed of n_1 equations $F_1(X) = 0, \ldots, F_{n_1} = 0$, n_2 inequalities $G_1(X) \geq 0, \ldots, G_{n_2})X) \geq 0$ and n_3 inequalities $H_1(X) \leq 0, \ldots, H_{n_3}(X) \leq 0$.

2.3.1 Mathematical background

2.3.1.1 Principle

Let $x_1, \ldots, x_m$ be the set of unknowns and let $\mathcal{I}_1 = \{(\underline{x_1^1}, \overline{x_1^1}), \ldots (\underline{x_m^1}, \overline{x_m^1})\}$ be the set of m intervals in which you are searching the solutions of the n equations $\mathcal{F}_1(x_1, \ldots, x_m) = 0, \ldots \mathcal{F}_n(x_1, \ldots, x_m) = 0$ (for the sake of simplicity we don't consider inequalities but the extension to inequalities is straightforward).

We will denote by F_i the interval value of $\mathcal{F}_i$ when this function is evaluated for the box $(\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_m}, \overline{x_m})$ of the unknowns while $F(\mathcal{I}_j)$ will denote the n-dimensional interval vector constituted of the F_i when the unknowns have the interval value defined by the set $\mathcal{I}_j$.

The algorithm will use a list of boxes $\mathcal{I}$ whose maximal size M is an input of the program. This list is initialized with $\mathcal{I}_1$. The number of $\mathcal{I}$ currently in the list is N and therefore at the start of the program $N = 1$. The algorithm will also use an accuracy on the variable ϵ and on the functions ϵ_F. The *norm* of a $\mathcal{I}$ is defined as:

$$\|\mathcal{I}^j\| = \mathrm{Max}(\overline{x_k^j} - \underline{x_k^j}) \quad \text{for } k \in [1, m]$$

The norm of the interval vector $F(\mathcal{I}_j)$ is defined as:

$$\uparrow F(\mathcal{I}_j) \uparrow = \mathrm{Max}(\overline{F_k(\mathcal{I}_j)} - \underline{F_k(\mathcal{I}_j)}) \quad \text{for } k \in [1, n]$$

The algorithm uses an index i and the result is a set $\mathcal{S}$ of interval vector $\mathcal{S}_k$ for the unknowns whose size is S. We assume that there is no F_k with k in $[1, n]$ such that $\overline{F(\mathcal{I}_1)} < 0$ or $\underline{F(\mathcal{I}_1)} > 0$, otherwise the equations have no solution in $\mathcal{I}_1$. Two lists of interval vectors $\mathcal{V}, \mathcal{W}$ whose size is 2^m will also be used. The algorithm is initialized with $i = 1, S = 1$ and proceed along the following steps:

1. if $i = N + 1$ return $S - 1$ and $\mathcal{S}$ and exit
2. bisect $\mathcal{I}_i$ which produce 2^m new interval vectors $\mathcal{V}_l$ and set $j = 1$
3. for $l = 1, \ldots, 2^m$
   (a) evaluate $F(\mathcal{V}_l)$
   (b) if it exist F_k with k in $[1, n]$ such that $\overline{F_k(\mathcal{V}_l)} < 0$ or $\underline{F_k(\mathcal{V}_l)} > 0$, then $l = l + 1$ and go to step 3
   (c) if $\|\mathcal{V}_l\| < \epsilon$ or $\uparrow F(\mathcal{V}_l) \uparrow < \epsilon_f$, then store $\mathcal{V}_l$ in $\mathcal{S}_S$, increment S and go to step 3
   (d) store $\mathcal{V}_l$ in $\mathcal{W}_j$, increment j and go to step 3
4. if $j = 1$ increment i and go to step 1
5. if $N + j - 2 > M$ return a failure code as there is no space available to store the new intervals

6. if $j > 1$ store one of the $\mathcal{W}$ in $\mathcal{I}_i$, the other $j-2$ at the end of $\mathcal{I}$, starting at position $N+1$. Add $j-2$ to N and go to step 1

Basically the algorithm just bisect the box until either their width is lower than ϵ or the width of the interval function is lower than ϵ_f (provided that there is enough space in the list to store the intervals). Then if all the intervals functions contain 0 we get a new solution, if one of them does not contain 0 there is no solution of the equations within the current box. A special case occurs when all the components of the box are reduced to a point, in which case a solution is obtained if the absolute value of the interval evaluation of the function is lower than ϵ_f.

Now three problems have to be dealt with:

1. how to choose the $\mathcal{W}$ which will be put in place of the $\mathcal{I}_i$ and in which order to store the other $\mathcal{W}$ at the end of the list?
2. can we improve the management of the bisection process in order to conclude the algorithm with a limited number M?
3. how do we distinguish distinct solutions ?

The first two problems will be addressed in the next section.

2.3.1.2 Managing the bisection and ordering

The second problem is solved in the following way: assume that at some step of the algorithm the bisection process leads to the creation of k $\mathcal{W}$ such that $N+k-2 > M$. As we have previously considered the $i-1$ elements of $\mathcal{I}$ we may use them as storage space. This means that we will store $\mathcal{I}_j, j \geq i$ at $\mathcal{I}_{j-i+1}$ thereby freeing $i-1$ elements. In that case the procedure will fail only if $N+k-i+1 > M$.

Now we have to manage the ordering of the $\mathcal{W}$. We have defined two types of order for a given set of boxes $\mathcal{I}$:

1. *maximum equation ordering*: the box are ordered along the value of $C = \text{Max}(|\overline{F_k(\mathcal{I})}|, |\underline{F_k(\mathcal{I})}|)$ for all k in [1,n]. The first box will have the lowest C.
2. *maximum middle-point equation ordering*: let C_i be the vector whose components are the middle points of the intervals $\mathcal{I}$. The box are ordered along the value of $C = \text{Max}(|\overline{F_k(C_i)}|, |\underline{F_k(C_i)}|)$ for all k in [1,n]. The first box will have the lowest C.

When adding the $\mathcal{W}$ we will substitute the $\mathcal{I}_i$ by the $\mathcal{W}$ having the lowest C while the others $\mathcal{W}$ will be added to the list $\mathcal{I}$ by increasing order of C. The purpose of these ordering is to try to consider first the box having the highest probability of containing a solution. This ordering may have an importance in the determination of the solution intervals (see for example section 2.3.5.2).

This method of managing the bisection is called the *Direct Storage* mode and is the default mode in `ALIAS`. But there is another mode, called the *Reverse Storage* mode. In this mode we still substitute the $\mathcal{I}_i$ by the $\mathcal{W}$ having the lowest C but instead of adding the remaining n $\mathcal{W}$ at the end of the list $\mathcal{I}$ we shift by n the boxes in the list, thereby freeing the storage of $\mathcal{I}_{i+1}, \ldots, \mathcal{I}_{i+1+n}$ which is used to store the remaining n $\mathcal{W}$. In other words we may consider the solving procedure as finding a leaves in a tree which are solutions of the problem: in the *Direct Storage* mode we may jump during the bisection from one branch of the tree to another while in the *Reverse storage* mode we examine all the leaves issued from a branch of the tree before examining the other branches of the tree. If we are looking for all the solutions the storage mode has no influence on the total number of boxes that will be examined. But the Reverse Storage mode may have two advantages:

- if we are looking for only one solution it may enable to find it more rapidly (but that is not compulsory, see section 2.3.5.4),
- as we are following one branch at a time we will consider very rapidly small box that either will lead to a solution or will be discarded thereby enabling to free some storage space. Hence the storage space available in the reverse mode will be in general higher than in the direct mode: a practical consequence is that a problem may not be solved with the direct mode due to problem in the storage while with the same amount of storage solutions will be obtained in the reverse mode.

To switch the storage mode see section 2.3.4.5 .We may also define a mixed strategy which is staring in the direct mode and then switching to the reverse mode when the storage becomes a problem (see section 8.3).

2.3.1.3 An alternative: the single bisection

A possibility to reduce the combinatorial explosion of the previous algorithm is to bisect not all the variables i.e. to use the *full bisection mode*, but only one of them (it must be noted that the algorithms in `ALIAS` will not accept a full bisection mode if the number of unknowns exceed 10). This may reduce the computation time as the number of function evaluation may be reduced. But the problem is to determine which variable should be bisected. All the solving algorithms of `ALIAS` may manage this single bisection by setting the flag `Single_Bisection` to a value different from 0. The value of this global variable indicates various bisection modes. Although the behavior of the mode may change according to the algorithm here are the possible modes for the general solving algorithm and the corresponding values for `Single_Bisection`:

- 1 : we just split the variable having the largest width (valid for all algorithms). Note however that it is still possible to order the bisection i.e. to split first a subset of the unknowns until their width is small (i.e. lower than `ALIAS_Accuracy`, then another subset and so on. This is obtained by setting flag `ALIAS_Ordered_Bisection` to 1 and defining an integer matrix `ALIAS_Order_Bisection` whose rows indicate the bisected subset and should end by 0. For example if this matrix has as rows [1,3,0],[2,4,5,0], then the algorithm will first bisect the unknowns 1 and 3 until their width is small, then the unknowns 2,4,5. If all unknowns indicated in the rows of the matrix have a small width, then the bisection algorithm revert to the normal behavior.

- 2: to determine the variable that will be bisected we use the following approach: we compute the order criteria for the two boxes P_1, P_2 that will result from the bisection of variable x_i and retain the lowest criteria c_i. The variable that will be bisected is the one that has the lowest c_i except if for at least one variable the interval evaluation of the function for P_1 or P_2 does not contain 0. In that case the variable that will be bisected is the one that verify the previous property and which has the lowest c_i among all the input intervals having the property. However to avoid bisecting over and over the same variable we use another test: let d_i be the width of the interval $[\underline{x_i}, \overline{x_i}]$ and d_{max} be the maximum of all the d_i. If $d_i/d_{max} < 0.1$ we don't consider the variable x_i as a possible bisection direction. It is also possible to mix this mode with mode 1. If the integer variable `ALIAS_RANDG` is set to a strictly positive value then `ALIAS_RANDG` bisection will be performed using mode 2 while the next bisection will be performed using mode 1 and the process will be repeated

- 3, 4 : similar to 1

- 5 : we use a *round-robin* mode i.e. each variable is bisected in turn (first x_1, then x_2 and so on) unless the width of the input intervals is less than the desired accuracy on the variable, in which case the bisected variable is the next one having a sufficient width (valid for all algorithms) The flag `ALIAS_Round_Robin` is used to indicate at each bisection which variable should be bisected.

- 6: we emulate the smear function (see section 2.4.1.3) with an estimation of the gradient based on finite difference (procedure `Select_Best_Direction_Grad`)

- 7: here again we use the flag `ALIAS_Ordered_Bisection` set to 1 and defining an integer matrix `ALIAS_Order_Bisection` whose rows indicates an order for bisecting the unknowns. The largest variable in the first row will be bisected first and so on until all the variables in a row have a width lower than `ALIAS_Accuracy`. We then proceed to the second row. As soon as all variables in all rows have a width lower than `ALIAS_Accuracy` we use the bisection 1.

For all general purpose solving procedures the number of the variable that has been bisected is available in the integer `ALIAS_Selected_For_Bisection`.

There is another mode called the *mixed bisection*: among the n variables we will bisect $m_1 < n$ variables, which will lead to 2^{m_1} new boxes. This mode is obtained by setting the global integer variable `ALIAS_Mixed_Bisection` to m_1. Whatever is the value of `Single_Bisection` we will order the variables according to their width and select the m_1 variables having the largest width.

2.3.1.4 Solutions and Distinct solutions

An interval will be considered as a solution for a function of the system in the following cases:

- for equations the maximal diameter of the intervals is less than a given threshold `epsilon` and the corresponding interval evaluation of the function contains 0 **or** the corresponding interval evaluation of the function has a diameter less than a given threshold `epsilonf` and the interval contains 0
- for inequalities $F(X) \leq 0$: the upper bound of the interval evaluation of the function is negative **or** the maximal diameter of the intervals is less than a given threshold `epsilon` and the corresponding interval evaluation of the function has at least a negative lower bound **or** the corresponding interval evaluation of the function has a diameter less than a given threshold `epsilonf` and the interval contains 0
- for inequalities $F(X) \geq 0$: the lower bound of the interval evaluation of the function is positive **or** the maximal diameter of the intervals is less than a given threshold `epsilon` and the corresponding interval evaluation of the function has at least a positive upper bound **or** the corresponding interval evaluation of the function has a diameter less than a given threshold `epsilonf` and the interval contains 0

A solution of the system is defined as a box such that the above conditions hold for each function of the system. Note that for systems having interval coefficients (which are indicated by setting the flag `ALIAS_Func_Has_Interval` to 1) a solution of a system will be obtained only if the inequalities are strictly verified.

Assume that two solutions $\mathcal{S}_\infty, \mathcal{S}_\in$ have been found with the algorithm. We will first consider the case where we have to solve a system of n equations in n unknowns, possible with additional inequality constraints. First we will check with the Miranda theorem (see section 3.1.5) if $\mathcal{S}_\infty, \mathcal{S}_\in$ include one (or more) solution(s). If both solutions are Miranda, then they will kept as solutions. If one of them is Miranda and other one is not Miranda we will consider the distance between the mid-point of $\mathcal{S}_\infty, \mathcal{S}_\in$: if this distance is lower than a given threshold we will keep as solution only the Miranda's one. If none of $\mathcal{S}_\infty, \mathcal{S}_\in$ is Miranda we keep these solutions, provided that their distance is greater than the threshold. Note that in that case these solutions may disappear if a Miranda solution is found later on such that the distance between these solutions and the Miranda's one is lower than the threshold.

In the other case the solution will be ranked according the chosen order and if a solution is at a distance from a solution with a better ranking lower than the threshold, then this solution will be discarded.

2.3.2 The 3B method

In addition to the classical bisection process all the solving algorithms in the `ALIAS` library may make use of another method called the *3B-consistency* approach [2].

Although its principle is fairly simple it is usually very efficient (but not always, see section 2.4.3.1). In this method we consider each variable x_i in turn and its range $[\underline{x_i}, \overline{x_i}]$. Let x_i^m be the middle point of this range. We will first calculate the interval evaluation of the functions in the system with the full ranges for the variable except for the variable i where the range will be $[\underline{x_i}, x_i^m]$. Clearly if one of the equations is not satisfied (i.e. its interval evaluation does not contain 0), then we may reduce the range of the variable i to $[x_i^m, \overline{x_i}]$. If this is not the case we will define a new x_i^m as the middle point of the interval $[\underline{x_i}, x_i^m]$ and repeat the process until either we have found an equation that is not satisfied (in which case the interval for the variable i will be reduced to $[x_i^m, \overline{x_i}]$) or the width of the interval $[\underline{x_i}, x_i^m]$ is lower than a given threshold δ. Using this process we will reduce the range for the variable i on the left side and we may clearly use a similar procedure to reduce it on the left side. The 3B procedure will be repeated if:

- 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 variable `ALIAS_Full3B` is set to 1 and the change in at least one variable is larger than `ALIAS_Full3B_Change`

For all the algorithms of `ALIAS` this method may be used by setting the flag `ALIAS_Use3B` to 1 or 2. In addition you will have to indicate for each variable a threshold δ and a maximal width for the range (if the width of the range is greater than this maximal value the method is not used). This is done through the `VECTOR` variables ALIAS_Delta3B and ALIAS_Max3B. The difference of behavior of the method if ALIAS_Use3B is set to 1 or 2 is the following:

- 1: let e be the value of ALIAS_Delta3B for the current variable which is in the range [a,b]. On the left side we will check if [a,a+e] may lead to no solution. If yes then the current value of the variable is [a+e,b]. We will start again but this time we will double the size of of the interval we will check i.e. we will test the elimination of [a+e,a+3e], then [a+3e,a+7e] and will stop as soon as the check on one interval fail. For example assume that the test for [a+3e,a+7e] fails, then the updated range for the variable will be [a+3e,b].

- 2: the procedure at the beginning is similar to the previous one but changes when the check fails. In the previous example after the failure for [a+3e,a+7e] we will start again to examine if interval with width e can be eliminated. Hence we will check [a+3e,a+4e], then [a+4e,a+6e] and so on. In consequence in this mode we will get as left bound for the interval the highest possible value A such that [A,A+e] cannot be eliminated. Clearly in that case the procedure will be more computer intensive but will produce better results.

A typical example for a problem with 25 unknowns will be:

```
ALIAS_Use3B=1;
Resize(ALIAS_Delta3B,25);Resize(ALIAS_Max3B,25);
for(i=1;i<=25;i++)
  {
    ALIAS_Delta3B(i)=0.1;ALIAS_Max3B(i)=7;
  }
```

which indicate that we will start using the 3B method as soon as the width of a range is lower than 7 and will stop it if we cannot improve the range by less than 0.1.

A drawback of the 3B method is that it may imply a large number of calls to the evaluation of the functions. The larger number of evaluation will be obtained by setting the ALIAS_Use3B to 2 and ALIAS_Full3B to 1 while the lowest number will be obtained if these values are 1 and 0. It is possible to specify that only a subset of the functions (the simplest) will be checked in the process. This is done with the global variable ALIAS_SubEq3B, an integer array whose size should be set to the number of functions and for which a value of 1 at position i indicates that the function i will be used in the 3B process while a value of 0 indicates that the function will not be used. For example:

```
Resize(ALIAS_SubEq3B,10);
Clear(ALIAS_SubEq3B);
ALIAS_SubEq3B(1)=1;
ALIAS_SubEq3B(2)=1;
```

indicates that only the two first functions will be used in the 3B process. If you are using your own solving procedure, then it is necessary to indicate that only part of the equations are used by setting the flag ALIAS_Use_SubEq3B to 1.

In some cases it may be interesting to try to use at least once the 3B method even if the width of the range is larger than ALIAS_Max3B. If the flag ALIAS_Always3B is set to 1, then the 3B will be used once to try to remove the left or right half interval of the variables.

If you are using also a simplification procedure (see section 2.3.3) you may avoid using this simplification procedure by setting the flag ALIAS_Use_Simp_In_3B to 0. You may also adapt the simplification procedure when it is called within the 3B method. For that purpose the flag ALIAS_Simp_3B is set to 1 instead of 0 when the simplification procedure is called within the 3B method. For some procedure if ALIAS_Use_Simp_In_3B is set to 2 then ALIAS_Simp_3B is set to 1 when the whole input is checked. But if ALIAS_Use_Simp_3B is set to a value larger than 2 then ALIAS_Simp_3B is set to 0.

Some methods allows to start the 3B method not by a small increment that is progressively increased but by a large increment (half the width of the interval) and to decrease it if it does not work. This is done by setting the flag ALIAS_Switch_3B to a value between 0 and 1: if the width of the current interval is lower than the width of the initial search domain multiplied by this flag, then a small increment is used otherwise a large increment is used.

When the routine that evaluate the expression uses the derivatives of the expression we may avoid to use these derivatives if the width of the ranges in the box are too large. This is obtained by assigning the size of the vector `ALIAS_Func_Grad` to the number of unknowns and assigning to the components of this vector to the maximal width for the ranges of the variables over which the derivatives will not be used: if there is a range with a width larger than its limits then no derivatives will be used.

Note also that the 3B-consistency is not the only one that can be used: see for example the ALIAS-Maple manual that implements another consistency test for equations which is called the *2B-consistency* or *Hull-consistency* in the procedure `HullConsistency` (similarly `HullIConsistency` implement it for inequalities). See also the section 2.17 for an ALIAS-C++ implementation of the 2B and section 11.4 for detailed calls to the 3B procedures.

2.3.3 Simplification procedure

Most of the procedures in `ALIAS` will accept as optional last argument the name of a *simplification procedure*: a user-supplied procedure that take as input the current box and proceed to some further reduction of the width of the box or even determine that there is no solution for these box, in which case it should return -1. Such procedure must be implemented as:

```
int Simp_Proc(INTERVAL_VECTOR & P)
```

where `P` is the current box. This procedure must return either -1 or any other integer. If a reduction of an interval is done within this procedure, then `P` must be updated accordingly.

This type of procedure allows the user to add information to the algorithm without having to add additional equations. The simplification procedure is applied on a box before the bisection and is used within the 3B method if this heuristic is applied.

Note that the Maple package associated to `ALIAS` allows in some cases to produce automatically the code for such procedure (see the ALIAS-Maple manual) and that section 2.17 presents a standard simplification procedure that may be used for almost any system of equations.

2.3.4 Implementation

The algorithm is implemented as:

```
int Solve_General_Interval(int m,int n,
     INTEGER_VECTOR Type_Eq,
     INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
     INTERVAL_VECTOR & TheDomain,
     int Order,int M,int Stop,
     double epsilon,double epsilonf,double Dist,
     INTERVAL_MATRIX & Solution,int Nb,
     int (* Simp_Proc)(INTERVAL_VECTOR &))
```

the arguments being:

- `m`: number of unknowns
- `n`: number of functions, see the note 2.3.4.1
- `Type_Eq`: type of the functions, see the note 2.3.4.2
- `IntervalFunction`: a function which return the interval vector evaluation of the functions, see the note 2.3.4.3
- `TheDomain`: box in which we are looking for solution of the system. A copy of the search domain is available in the global variable `ALIAS_Init_Domain`
- `Order`: the type of order which is used to store the intervals created during the bisection process. This order may be either `MAX_FUNCTION_ORDER` or `MAX_MIDDLE_FUNCTION_ORDER`. See the note on the order 2.3.4.4.
- `M`: the maximum number of boxes which may be stored. See the note 2.3.4.5

- `Stop`: the possible values are 0,1,2
  - 0: the algorithm will look for every solution in `TheDomain`
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found
- `epsilon`: the maximal width of the solution intervals, see the note 2.3.4.6
- `epsilonf`: the maximal width of the function intervals for a solution, see the note 2.3.4.6
- `Dist`: minimal distance between the middle point of two interval solutions, see the note 2.3.4.7
- `Solution`: an interval matrix of size (`Nb`,`m`) which will contained the solution intervals. This list is sorted using the order specified by `Order`
- `Nb`: the maximal number of solution which will be returned by the algorithm
- `Simp_Proc`: a user-supplied procedure that take as input the current box and proceed to some further reduction of the width of the box or even determine that there is no solution for this box, in which case it should return `-1`. Remember also that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2).

Note that the following arguments may be omitted:

- `Type_Eq`: in that case all the functions will supposed to be equations.
- `Simp_Proc`: no simplification procedure is provided by the user

2.3.4.1 Number of unknowns and functions

The only constraint on `n`,`m` is that they should be strictly positive. So the algorithm is able to deal with under-constrained or over-constrained systems.

2.3.4.2 Type of the functions

The i-th value in the array of n integers `Type_Eq` enable one to indicate if function F_i is an equation or an inequality:

- `Type_Eq`(i)=0 : F_i must verify $F_i(X) = 0$
- `Type_Eq`(i)=1 : F_i must verify $F_i(X) \geq 0$
- `Type_Eq`(i)=-1 : F_i must verify $F_i(X) \leq 0$

For all the general solving algorithms the integer `ALIAS_Pure_Equation` is set to the number of equations in the constraints list.

2.3.4.3 Interval Function

The user must provide a function which will compute the function intervals of the functions for a given box. When designing `ALIAS` we have determined that to be efficient we need a procedure that allow to calculate the interval evaluation of all the functions or only a subgroup of them in order to avoid unnecessary calculations. Hence the syntax of this procedure is:

```
INTERVAL_VECTOR IntervalFunction (int l1,int l2,INTERVAL_VECTOR & x)
```

- `x`: a m dimensional interval vector which define the intervals for the unknowns
- `l1,l2`: the function must be able to return the interval value of the functions `l1` to `l2`. The first function has number 1, the last `m`. So if `l1`=`l2`=1 the function should return an interval vector whose only the first component has been computed.

This function should be written using the BIAS/Profil rules. If you have equations and inequalities in the system you **must** define first the equations and then the inequalities.

The efficiency of the algorithm is heavily dependent on the way this procedure is written. Two factors are to be considered:

- efficiency of the evaluation
- sharp bound on the evaluation

Efficiency will enable to decrease the computation time of the evaluation. Let consider for example the following system:

$$x^2+y^2-50=0$$
$$x^2-20\,x+8\,x\cos(\theta)+90-80\,\cos(\theta)+y^2+8\,y\sin(\theta)=0$$
$$x^2-6\,x+4\,x\cos(\theta)-4\,x\sin(\theta)+92-52\,\cos(\theta)-28\,\sin(\theta)+y^2-20\,y+4\,y\sin(\theta)+4\,y\cos(\theta)=0$$

The evaluation function may be written as:

```
e1 = Sqr(x)+Sqr(y)-50.0 ;
e2 =Sqr(x)-20.0*x+8.0*x*Cos(teta)+90.0-80.0*Cos(teta)+Sqr(y)+8.0*y*Sin(teta);
e3 =Sqr(X)-6.0*x+4.0*x*Cos(teta)-4.0*x*Sin(teta)+92.0-52.0*Cos(teta)-28.0*
      Sin(teta)+SQR(Y)-20.0*y+4.0*y*Sin(teta)+4.0*y*Cos(teta);
```

or, using temporary variables:

```
t1 = Sqr(x);
t2 = Sqr(y);
t5 = Cos(teta);
t6 = x*t5;
t9 = Sin(teta);
t10 = y*t9;
e1 = t1+t2-50.0;
e2 = t1-20.0*x+8.0*t6+90.0-80.0*t5+t2+8.0*t10;
e3 = t1-6.0*x+4.0*t6-4.0*x*t9+92.0-52.0*t5-28.0*t9+t2-20.0*y+4.0*t10+4.0*y*t5;
```

the second manner is more efficient as the intervals $\sin\theta, \cos\theta, x^2, y^2,\ x\cos\theta,\ y\sin\theta$ are evaluated only once instead of 3 or 2 in the first evaluation. Note also that for speeding up the computation it may be interesting to declare the variables `t1, t2, t5, t6, t9, t10` as global to avoid having to create a new interval data structure at each call of the evaluation function.

The second point is the sharpness of the evaluation. Let consider the polynomial x^2-x. If the variable lie in the interval [0,1] the evaluation will lead to the interval [-1,1]. The same polynomial may we written in Horner form as $x(x-1)$ the function being then evaluated as [-1,0]. Now suppose that x lie in [0.8,1.1]. The initial polynomial will be evaluated as [-0.46,0.41] while in Horner form the evaluation leads to [-0.22,0.11]. But this polynomial may also be written as $(x-1)^2+x-1$ (which is the *centered form* at 1) whose evaluation leads to [-0.2,0.14] which has a sharper lower bound than in the Horner form (note that Horner form is very efficient for the evaluation of a polynomial but do not lead always to the sharpest evaluation of the bounds on the polynomial although this is some time mentioned in the literature). Unfortunately there is no known method which enable to determine what is the best way to express a given function in order to get the sharpest possible bounds. For complex expression you may use the procedures `MinimalCout` or `Code` of ALIAS-Maple that try to produce the less costly formulation of a given expression.

Another problem is the cost of the tests which are necessary to determine if the interval evaluation of one of the function does not include 0. Indeed let us assume that we have 40 equations and 7 unknowns and that we are considering a box such that the function interval all contain 0. When testing the functions we may either evaluate all the functions with one procedure call (with the risk of performing useless evaluations e.g. if the interval evaluation of the first equations does not contain 0) or evaluate the functions one after the other (at a cost of 40 procedure calls but avoiding useless equation evaluations). The best way balances the cost of procedure calls compared to the cost of equation evaluations. By default we are evaluating all the functions in one step but by setting the variable `Interval_Evaluate_Equation_Alone` to 1 the program will evaluate the functions one after the other.

A last problem is the interval valuation of the equations. Indeed you may remember that some expression may not be evaluated for some ranges for the unknowns (see section 2.1.1.3). If such problem may occur a solution is to include into this procedure a test before each expression evaluation that verify if the expression is interval-valuable. If not two cases may occur:

- the expression will never be interval-valuated whatever is the value of one of the unknown in its range (e.g. the expression involves `ArcSin(x)` and the range for `x` is [-4,-3])
- the expression may be evaluated for some values of the unknowns in their range (e.g. the expression involves `Sqrt(x)` while the range for `x` is [-3,10])

In the first case the interval evaluation of the expression should be set to an interval which does not include 0 (hence the algorithm will discard the box). In the second case the best strategy seems to be to set the interval evaluation of the expression to a very large interval that includes 0 (e.g. [-1e8,1e8]). Note that some filtering strategy such as the one described in section 2.17 may help to avoid some of these problems (in the example this strategy will have determined that `Sqrt(x)` is interval-valuable only for `x` in [0,10] and will have automatically set the range for `x` to this value).

2.3.4.4 The order

Basically the algorithm just bisect the box `TestDomain` until one of the criteria described in 2.3.4.6 is satisfied. The boxes resulting from the bisection process are stored in a list and the boxes in the list are treated sequentially. If we are looking only for one solution of the equation or for the first `Nb` solutions of a system (see the `Stop` variable) it is important to store the new boxes in the list in an order which ensure that we will treat first the boxes having the highest probability of containing a solution. Two types of ordering may be used, see section 2.3.1.2, indicated by the flag `MAX_FUNCTION_ORDER` or `MAX_MIDDLE_FUNCTION_ORDER`.

Note that if we are looking for all the solutions of the system the order has still an importance: although all the boxes of the list will be treated the order define how close solution intervals will be distinguished (see for example section 2.3.5.2).

2.3.4.5 Storage

The boxes generated by the bisection process are stored in an interval matrix:

```
Box_Solve_General_Interval(M,m)
```

The algorithm try to manage the storage in order to solve the problem with the given number `M`. As seen in section 2.3.1.2 two storage modes are available, the *Direct Storage* and the *Reverse Storage* modes, which are obtained by setting the global variable `Reverse_Storage` to 0 (the default value) or at least to the number of unknowns plus 1. See also section 8.3 to use a mixed strategy between the direct and reverse mode.

For both modes the algorithm will first run until the bisection of the current box leads to a total number of boxes which exceed the allowed total number. It will then delete the boxes in the list which have been already bisected, thereby freeing some storage space (usually larger for the reverse mode than for the direct mode) and will start again.

If this is not sufficient the algorithm will consider each box in the list and determine if the bisection process applied on the box does create any new boxes otherwise the box is deleted from the list. Note that this procedure is computer intensive and constitute a "last ditch" effort to free some storage space. You can disable this feature by setting the integer variable `Enable_Delete_Fast_Interval` to 0. If the storage space freed by this method is not sufficient the algorithm will exit with a failure return.

If `epsilonf`=0, `epsilon`=ϵ and f is the largest width of the intervals in `TestDomain`, then the number of boxes that will be considered in the direct mode is M with, in the worst case:

$$M = 2^{\mathrm{Sup}(\frac{f}{2\epsilon})+1} - 1 \tag{2.1}$$

where $\mathrm{Sup}(\frac{f}{2\epsilon})$ is the largest integer greater than $f/2\epsilon$. In the direct storage mode the storage space N will be in the worst case:

$$N = 2^{\mathrm{Sup}(\frac{f}{2\epsilon})} - 1 \tag{2.2}$$

In the reverse storage mode the storage space is only:

$$N = \frac{log(\frac{f}{2\epsilon})}{log(2)} + 1 \tag{2.3}$$

Note that with the reverse storage mode, storage is not really a problem. For example if the width of the initial box is 1000 and the accuracy 10^{-10}, then the necessary storage space is only 44. Thus only the computation time or the conditioning of the functions may lead to a failure of the algorithm. If `epsilonf` is not equal to 0 the size of the storage cannot be estimated.

If the procedure has to be used more than once it is possible to speed up the computation by allocating the storage space before calling the procedure. Then you may indicate that the storage space has been allocated beforehand by indicating a negative value for `M`, the number of boxes being given by the absolute value of `M`.

Note also that the bisection process applied only to one variable may lead to a better estimation of the roots of the system if the algorithm stops when the accuracy required on the variable is reached: indeed, compared to the standard algorithm, one (or more) of the variable may have been individually split before reaching the step where a full bisection will lead to a solution (see the example in section 2.4.3.2).

Note also a specific use of `ALIAS_RANDG`: if this integer is not set to 0, then every `ALIAS_RANDG` iteration the algorithm will put the box having the largest width as current box, except if the number of boxes remaining to be processed is greater than half the total number of available boxes.

2.3.4.6 Accuracy

Two criterion are used to determine if a box possibly includes a solution of the system:

- the largest width of the components of the box is lower than `epsilon` and the functions intervals for this box all contain 0
- the largest width of the function intervals is lower than `epsilonf` and they contain all 0. You must be aware that this test is only used if there is no inequality in the system. **In that case it is compulsory to have an `epsilon` not equal to 0 otherwise the procedure may lead to an infinite loop**.

If we use only the first criteria (i.e. we put `epsilonf`=0) the largest width of the solution intervals will be `epsilon`. A consequence is that the unknowns should be normalized in order that all the intervals in the `TestDomain` have roughly the same width.

If we use only the second criteria the width of the solution intervals cannot be determined and the functions should be roughly normalized (see the example in section 15.1.3 for the importance of the conditioning).

2.3.4.7 Distinct solutions

Two solution intervals will be assumed to contain distinct solutions if the minimal distance between the middle point of all the intervals is greater than the threshold `Dist`.

2.3.4.8 Return code

The procedure will return an integer k

- $k \geq 0$: number of solutions
- $k = -1$: the size of the storage is too low (possible solutions: increase `M`, or use the 3B method, or use the reverse storage mode or the single bisection mode)
- $k = -2$: `m` or `n` is not strictly positive
- $k = -3$: `Order` is not 0 or 1

- $k = -4$: one of the function in the system has not a type 0, -1 or 1 (i.e. it's not an equation, neither inequality $F \leq 0$ or an inequality $F \geq 0$)
- $k = -5$: we are in the optimization mode and more than one functions are expressions to be optimized (see the Optimization chapter)
- $k = -100$: in the mixed bisection mode the number of variables that will be bisected is larger than the number of unknowns
- $k = -200$: one of the value of `ALIAS_Delta3B` or `ALIAS_Max3B` is negative or 0
- $k = -300$: one of the value of `ALIAS_SubEq3B` is not 0 or 1
- $k = -400$: although `ALIAS_SubEq3B` has as size the number of equations none of its components is 1
- $k = -500$: `ALIAS_ND` is different from 0 (i.e. we are dealing with a non-0 dimensional problem, see the corresponding chapter) and the name of the result file has not been specified
- $k = -1000$: the value of the flag `Single_Bisection` is not correct
- $k = -3000$: we use the full bisection mode and the problem has more than 10 unknowns

2.3.4.9 Debugging

If the algorithm fail some debugging options are provided. The integer variable `Debug_Level_Solve_General_Interval` indicate which level of debug is used:

- 0: no debug (the default value)
- 1: during the process are printed on the standard output: the index of the current box, the total number of boxes and the number of remaining boxes together with the current number of solutions
- 2 : same as 1 but the intervals of the current box are also printed and when it is split the new boxes are printed together with their function intervals

2.3.5 Examples and Troubleshooting

2.3.5.1 Example 1

We will present first a very silly example of system in the three unknowns $x(1), x(2), x(3)$:

$$x(1)^2 + 2x(1) + 1 = 0$$
$$x(2)^2 + 2x(2) + 1 = 0$$
$$x(3)^2 + 2x(3) + 1 = 0$$

Clearly this system has the unique solution (-1,-1,-1). We choose to define the `TestDomain` as the interval [-2,0] for all three unknowns. So we define a function which specify the `TestDomain`:

```
VOID SetTestDomain (INTERVAL_VECTOR & x)
{
  Resize (x, 3);
  x(1) = Hull (-2.0,0.0);
  x(2) = Hull (-2.0,0.0);
  x(3) = Hull (-2.0,0.0);
}
```

The we have to define the `IntervalFunction`:

```
INTERVAL_VECTOR IntervalTestFunction (int l1,int l2,INTERVAL_VECTOR & x)
// interval valued functions. The input are intervals for the
//variables and the output is intervals on the functions
//x are the input variables and xx the function intervals
{
INTERVAL_VECTOR xx(3);

if(l1==1)  xx(1)=x(1)*(x(1)+2)+1;
if(l1<=2 && l2>=2)  xx(2)=x(2)*(x(2)+2)+1;
if(l2==3)  xx(3)=x(3)*(x(3)+2)+1;
  return xx;
}
```

This function returns the interval vector `xx` which will contain the value of the function from `l1` to `l2` for the box `x`. Note that the initial functions have been written in Horner form (or "nested" form) which may lead to a sharper estimation of the function intervals.

The main program may be written as:

```
INT main()
{
int Num; //number of solution
INTERVAL_MATRIX SolutionList(1,3);//the list of solutions
INTERVAL_VECTOR TestDomain;//the input intervals for the variable

//We set the value of the variable intervals
SetTestDomain (TestDomain);

//let's solve....
Num=Solve_General_Interval(3,3,IntervalTestFunction,TestDomain,
      MAX_FUNCTION_ORDER,50000,1,0.001,0.0,0.1,SolutionList,1);
//too much intervals have been created, this is a failure
if(Num== -1)cout << "The procedure has failed (too many iterations)"<<endl;
return 0;
}
```

This main program will stop as soon as one solution has been found. We set `epsilonf` to 0 and `epsilon` to 0.001 which mean that the maximal width of the interval solution will be lower than 0.001. The chosen order is the maximum equation ordering. The number of boxes should not be larger than 50000 and the distance between the distinct solutions must be greater than 0.1.

Running this program will provide the solution interval [-1.000977,-1] for all three variables and uses 71 boxes. The result will have been similar if we have chosen the maximum middle-point equation ordering.

On the other hand if we have `epsilon` to 0 and `epsilonf` to 0.001 the algorithm find the solution interval [-1.00048828125,-1] and use 78 boxes.

Now let's look at a more complete test program which enable to test the various options of the procedure.

```
INT main()
{
int Iterations;//maximal size of the storage
int Dimension,Dimension_Eq; // size of the system
int Num,i,j,order,precision,Stop;
// accuracy of the solution either on the function  or on the variable
double Accuracy,Accuracy_Variable,Diff_Sol;
INTERVAL_MATRIX SolutionList(200,3);//the list of solutions
INTERVAL_VECTOR TestDomain;//the input intervals for the variable
INTERVAL_VECTOR F(3),P(3);

//We set the number of equations and unknowns and the value of the variable intervals
Dimension_Eq=Dimension=3;
SetTestDomain (TestDomain);

cerr << "Number of iteration = "; cin >> Iterations;
cerr << "Accuracy on Function = "; cin >> Accuracy;
cerr << "Accuracy on Variable = "; cin >> Accuracy_Variable;cerr << "Order (0,1)"; cin >>order;
cerr << "Stop at first solutions (0,1,2):";cin>>Stop_First_Sol;
cerr << "Separation between distincts solutions:";cin>> Diff_Sol;

//let's solve....
Num=Solve_General_Interval(Dimension,Dimension_Eq,IntervalTestFunction,TestDomain,order,Iterations,Stop,
      Accuracy_Variable,Accuracy,Diff_Sol,SolutionList,1);
//too much intervals have been created, this is a failure
if(Num== -1){cout<<"Procedure has failed (too many iterations)"<<endl;return -1;}
cout << Num << " solution(s)" << endl;

for(i=1;i<=Num;i++)
  {
    cout << "solution " << i <<endl;
    cout << "x(1)=" << SolutionList(i,1) << endl;
    cout << "x(2)=" << SolutionList(i,2) << endl;
    cout << "x(3)=" << SolutionList(i,3) << endl;
    cout << "Function value at this point" <<endl;
    for(j=1;j<=3;j++)F(j)=SolutionList(i,j);
    cout << IntervalTestFunction(1,Dimension_Eq,F) <<endl;
    cout << "Function value at middle interval" <<endl;
    for(j=1;j<=3;j++)P(j)=Mid(SolutionList(i,j));
    F=IntervalTestFunction(1,Dimension_Eq,P);
    for(j=1;j<=3;j++)cout << Sup(F(j)) << endl;
  }
  return 0;
}
```

This program is basically similar to the previous one except that it enable to define interactively the order, `M`, `Stop`, `epsilon`, `epsilonf`,`Dist`. Then it print the solution together with the function interval for the solution interval and the value of the functions at the middle point of the solution interval. Let's test the algorithm to find all he solution (`Stop`=0) with `epsilonf`=0 and `epsilon`=0.001. The algorithm will fail: indeed let's compute the maximal storage that we may need using the formula (2.1). We end up with the number 2^{6000} which is indeed a very large number.... But (2.1) is only an upper bound for the storage. For example if we have used `epsilon`=0.01 in the previous formula we will find that `M`=2^{600} although the algorithm converge toward [-1,-0.992188] while using only 5816 boxes.

Although academic this system shows several properties of interval analysis. If we set `epsilon`=`epsilonf`=1e-6, which is the standard setting of ALIAS-Maple, the algorithm will run for a very long time before finding the solution of the system. Even using the 3B method, section 2.3.2 and the 2B method (section 2.17) the computation time, although improved, will still be very large. Now if we write the system as

$$(x(1)+1)^2 = 0$$
$$(x(2)+1)^2 = 0$$
$$(x(3)+1)^2 = 0$$

the system will be solved with only 8 boxes, with 2 boxes if we use the 3B method and without any bisection if we use the 2B method. This shows clearly the importance of writing the equations in the most compact form.

2.3.5.2 Example 2

The problem we want to solve is presented in section 15.1.1. We consider a system of three equations in the unknowns x, y, θ:

$$x^2 + y^2 - 50 = 0$$
$$x^2 - 20\,x + 8\,x\cos(\theta) + 90 - 80\,\cos(\theta) + y^2 + 8\,y\sin(\theta) = 0$$
$$x^2 - 6\,x + 4\,x\cos(\theta) - 4\,x\sin(\theta) + 92 - 52\,\cos(\theta) - 28\,\sin(\theta) + y^2 - 20\,y + 4\,y\sin(\theta) + 4\,y\cos(\theta) = 0$$

which admit the two solutions:

$$(5, 5, 0) \qquad (3.369707132, 6.216516219, -0.806783438)$$

By looking at the geometry of the problem it is easy to establish a rough `TestDomain`:

```
VOID SetTestDomain (INTERVAL_VECTOR & x)
{
  Resize (x, 3);
  x(1) = Hull (0.9,7.1);
  x(2) = Hull (2.1,7.1);
  x(3) = Hull (-Constant::Pi,Constant::Pi);
}
```

and to determine that the maximum number of real solution is 6. The `IntervalFunction` is written as:

```
INTERVAL_VECTOR IntervalTestFunction (int l1,int l2,INTERVAL_VECTOR & in)
{
INTERVAL_VECTOR xx(3);
if(l1==1)xx(1)=in(1)*in(1)+in(2)*in(2)-50.0;
if(l1<=2 && l2>=2)
  xx(2)=-80.0*Cos(in(3))+90.0+(8.0*Sin(in(3))+in(2))*in(2)+(-20.0+8.0*Cos(in(3))+in(1))*in(1);
if(l2==3)
 xx(3)=92.0-52.0*Cos(in(3))-28.0*Sin(in(3))+(-20.0+4.0*Sin(in(3))+
       4.0*Cos(in(3))+in(2))*in(2)+(-4.0*Sin(in(3))-6.0+4.0*Cos(in(3))+in(1))*in(1);
  return xx;
}
```

and we may use the same main program as in the previous example (the name of this program is `Test_Solve_General1`).

Let's assume that we set `epsilonf` to 0 and `epsilon` to 0.01 while looking at all the solutions (`Stop`=0), using the maximum equation ordering and setting `Dist` to 0.1. The algorithm provide the following solutions,using 684 boxes:

$$x = [4.99297, 4.99902] \quad y = [5.00527, 5.01016] \quad \theta = [-0.00613592, 0]$$
$$x = [3.36426, 3.37031] \quad y = [6.21133, 6.21621] \quad \theta = [-0.809942, -0.803806]$$

We notice that indeed none of the roots are contained in the solution intervals. If we use the maximum middle-point equation ordering the algorithm provide the solution intervals, using 684 boxes:

$$x = [3.36426, 3.37031] \quad y = [6.21621, 6.22109] \quad \theta = [-0.809942, -0.803806]$$
$$x = [5.00508, 5.01113] \quad y = [4.99063, 4.99551] \quad \theta = [0, 0.00613592]$$

which still does not contain the root (5,5,0) (but contain one of the root which show the importance of the ordering). Let's look at what is happening by setting the debug flag `Debug_Level_Solve_General_Interval` to 2 (see section 2.3.4.9). At some point of the process the algorithm has determined four different solution intervals:

Solution1	$[4.9990234375, 5.005078125]$	$[4.990625, 4.9955078125]$	$[0, 0.006135923151542]$
Solution2	$[4.9990234375, 5.005078125]$	$[4.9955078125, 5.000390625]$	$[0, 0.006135923151542]$
Solution3	$[5.005078125, 5.0111328125]$	$[4.990625, 4.9955078125]$	$[0, 0.006135923151542]$
Solution4	$[3.3642578125, 3.3703125]$	$[6.211328125, 6.2162109375]$	$[-0.809941856, -0.803805932852]$

the criteria C for the ordering being:

0.300481441333159	0.329822553021982
0.293359098885522	0.416913915955262

Clearly solution 3 has the lowest criteria and will therefore be stored as the first solution. Then solution 1 will be considered: but the distance between the middle point of solution 3 and 1 is lower than `Dist` and therefore solution 1 will not be retained. The solution 2 will be considered but for the same reason than for solution 1 this solution will not been retained. Finally solution 4 will be considered and it spite of his index being the worse this solution will be retained as its distance to solution 3 is greater than `Dist`.

Note that if the single bisection is activated and setting the flag `Single_Bisection` to 1 we find the two roots for `epsilonf` to 0 and `epsilon` to 0.01 with 650 boxes using the maximum equation ordering.

We may also illustrate on this example how to deal with inequalities. Assume now that we want to deal with the same system but also with the inequality $xy - 22 \leq 0$. We modify the `IntervalTestFunction` as:

```
INTERVAL_VECTOR IntervalTestFunction (int l1,int l2,INTERVAL_VECTOR & in)
{
INTERVAL x,y,teta;
INTERVAL_VECTOR xx(4);

x=in(1);y=in(2);teta=in(3);
if(l1==1)xx(1)=x*x+y*y-50.0;
if(l1<=2 && l2>=2)
  xx(2)=-80.0*Cos(teta)+90.0+(8.0*Sin(teta)+y)*y+(-20.0+8.0*Cos(teta)+x)*x;
if(l1<=3 && l2>=3)
 xx(3)=92.0-52.0*Cos(teta)-28.0*Sin(teta)+(-20.0+4.0*Sin(teta)+
       4.0*Cos(teta)+y)*y+(-4.0*Sin(teta)-6.0+4.0*Cos(teta)+x)*x;
if(l2==4)
 xx(4)=x*y-22.;
  return xx;
}
```

Part of the main program will be:

```
Type(1)=0;Type(2)=0;Type(3)=0;Type(4)=-1;
Num=Solve_General_Interval(3,4,Type,IntervalTestFunction,TestDomain,order,
Iterations,Stop_First_Sol,Accuracy_Variable,
Accuracy,Diff_Sol,SolutionList,6);
```

Here `Type(4)=-1;` indicates that the fourth function is an inequality of the type $F_i(X) \leq 0$. If we have to deal with the constraint $xy - 22 \geq 0$ then we will use `Type(4)=1;`.

2.3.5.3 Example 3

This example is derived from example 2. We notice that in the three functions of example 2 the second degree terms of x, y are for all functions $x^2 + y^2$. Thus by subtracting the first function to the second and third we get a linear system in x, y. This system is solved and the value of x, y are substituted in the first function. We get thus a system of one equation in the unknown θ (see section 15.1.2). The roots of this equation are 0,-0.806783438. The test program is `Test_Solve_General2`. The `IntervalFunction` is written as:

```
INTERVAL_VECTOR IntervalTestFunction (int l1,int l2,INTERVAL_VECTOR & in)
{
INTERVAL_VECTOR xx(1);
xx(1)=11092.0+(-25912.0+(19660.0-4840.0*Cos(in(1)))*Cos(in(1)))*Cos(in(1))+(
-508.0+(3788.0-1600.0*Cos(in(1)))*Cos(in(1)))*Sin(in(1));
return xx;
}
```

This program is implemented under the name `Test_Solve_General2`. With `epsilonf`=0 and `epsilon`=0.001 we get the solution intervals, using 32 boxes:

$$\theta = [-0.0007669904, 0] \quad \theta = [-0.8068739, -0.8061069]$$

for whatever order. If we use `epsilon`=0 and `epsilonf`=0.1 we get, using 50 boxes:

$$\theta = [-0.806784012741056, -0.806781016684830]$$
$$\theta = [-4.793689962142628e-05, 0]$$

In both cases the solution intervals contain the roots of the equation.

2.3.5.4 Example 4

In this example (see section 15.1.3) we deal with a complex problem of three equations in three unknowns ψ, θ, ϕ. We are looking for a solution in the domain:

$$[4.537856054, 4.886921908], [1.570796327, 1.745329252], [0.6981317008, 0.8726646262]$$

The system has a solution which is approximately:

$$4.6616603883, 1.70089818026, 0.86938888189$$

This problem is extremely ill conditioned as for the `TestDomain` the functions intervals are:

$$[-1.45096e+08, 1.32527e+08]; [-38293.3, 29151.5]; [-36389.1, 27705.7]$$

This program is implemented under the name `Test_Solve_General`. With `espsilonf`=0 and `epsilon`=0.001 and if we stop at the first solution we find with the maximum equation ordering:

$$\psi = [4.664665, 4.665347] \quad \theta = [1.7034, 1.703741] \quad \phi = [0.8706193, 0.8709602]$$

with 531 boxes. We may also mention the following remarks:

- we get no improvement with the single bisection mode as we need 2435 boxes to find the first solution,
- using the Reverse Storage mode does not lead to any improvement for finding the first root: in this mode we need 5587 boxes to get the first solution,

With the maximum middle-point equation ordering we find:

$$\psi = [4.665347, 4.666029] \quad \theta = [1.701355, 1.701696] \quad \phi = [0.8709602, 0.8713011]$$

with 203 boxes. The importance of normalizing the functions appears if we use `epsilonf`=0.1 and `epsilon`=0. If we stop at the first solution we find:

$$\psi = [4.661660388259656, 4.661660388340929]$$
$$\theta = [1.700898180243437, 1.700898180284073]$$
$$\phi = [0.869388881899751, 0.869388881940387]$$

while if we divide the first function by 1000 we find:

$$\psi = [4.661658091884636, 4.661658424779772]$$
$$\theta = [1.700898403947993, 1.700898570395561]$$
$$\phi = [0.869388105618527, 0.869388272066095]$$

in four time less computation time.

2.3.5.5 General comments

The advantages of the proposed algorithm is that it is easy to use and implement for a fast check. For sharp system it may provide quickly solutions with a reasonable accuracy. The drawback is that it may provide solutions intervals which does not contain roots or, worse, miss some roots if `Dist` is not set to 0 (see section 2.3.5.2).

This algorithm may be used also for analysis: if we have to solve numerous systems we may use this algorithm with a low `M` in order to fast check if the current system may have some real roots, in which case we may consider using a more sophisticated algorithm.

This solving algorithm may be tested on-line on the web page `ALIAS-on-line` that can be found on the COPRIN web page.

2.4 General purpose solving algorithm with Jacobian

2.4.1 Mathematical background

Assume now that we are able to compute the jacobian matrix of the system of functions. We will use this jacobian for improving the evaluation of the function intervals using two approaches:

1. use the monotonicity of the function
2. use the gradient for the evaluation of the function intervals

Note also that this method may be used to solve a system composed of n_1 equations $F_1(X) = 0, \ldots, F_{n_1} = 0$, n_2 inequalities $G_1(X) \geq 0, \ldots, G_{n_2})X) \geq 0$ and n_3 inequalities $H_1(X) \leq 0, \ldots, H_{n_3}(X) \leq 0$. Note also that not all the function must be differentiable to use this procedure: only one of them is sufficient. In that case you will have however to set special values in the gradient function (see 2.4.2.2).

A notable difference with the previous procedure is that we use Moore theorem (see section 3.1.1) to determine if a unique solution exists in a given box, in which case we use Krawczyk method for determining this solution (see section 2.10). Therefore if the algorithm proposes as solution a point instead of a range this imply that this solution has been obtained as the result of Moore theorem. Note however that getting a range for a solution instead of a point does not always imply that we have a singular solution. For example it may happen that the solution is exactly at one extremity of a box (see example in section 2.4.3.2) which a case that our algorithm does not handle very well. A local analysis of the solution should however confirm quickly if the solution is indeed singular.

In addition we use also the inflation method presented in section 3.1.6 to increase the width of the box in which we may guarantee that there is a unique solution.

Hence this algorithm allows to determine *exact* solutions in the sense that we determine boxes that contains a unique solution and provides a guaranteed numerical scheme that allows for the calculation of the solution.

In the same way we use the Hansen-Sengupta version of the interval Newton method to improve the boxes (see [21]). Note that an improved interval Newton that may benefit from the structure of the system is available (see section 3.1.4).

This algorithms allows also to determines the solutions of non-0 dimensional system, see section 2.2.

2.4.1.1 Using the monotonicity

For a given box we will compute the jacobian matrix using interval analysis. Each row j of this interval matrix enable to get some information of the corresponding function F_j.

- if the i-th column of the j-th row is an interval which is strictly negative or strictly positive, then F_j is monotonic with respect to the unknowns x_i
- if the i-th column of the j-th row is equal to 0, then function F_j does not depend on the variable x_i

In the first case the minimal and maximal value of F_j will be obtained either for $x_i = \underline{x_i}$ or $x_i = \overline{x_i}$ and we are able to define the value of x_i to get successively the minimal and maximal value as we know the sign of the gradient. But this procedure has to be implemented recursively. Indeed we have previously computed the jacobian matrix for $x_i = (\underline{x_i}, \overline{x_i})$ but now x_i have a fixed value: hence a component J_{jk} of the j-th row which for $x_i = (\underline{x_i}, \overline{x_i})$ was such that $\underline{J_{jk}} < 0$ and $\overline{J_{jk}} > 0$ may now be a strictly positive or negative intervals. Consequently the minimal and maximal value will be obtained for some combination of x_i, x_k in the two sets $\{\underline{x_i}, \overline{x_i}\}$ and $\{\underline{x_k}, \overline{x_k}\}$. Bus as x_k has now a fixed value some other component of J_k may become strictly negative or positive...

The algorithm for computing a sharper evaluation of F_j is:

$(\underline{F_j}, \overline{F_j})$=`Evaluate`$F_j((\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_m}, \overline{x_m}))$

1. compute $J_j((\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_m}, \overline{x_m}))$
2. let l_1 be the number of components of J_j such that $\underline{J_{jk}} > 0$ or $\overline{J_{jk}} < 0$ and let $x_l, \ldots, x_p$ be the variables for which this occur
3. if $l_1 > 0$
   `loop:` for all combination of $x_l, \ldots, x_p$ in the set $\{\underline{x_l}, \overline{x_l}, \ldots, \underline{x_p}, \overline{x_p}\}$:
   - if $l_1 < m$
     - compute $J_j((\underline{x_1}, \overline{x_1}), \ldots, x_l, \ldots, x_p, \ldots, (\underline{x_m}, \overline{x_m}))$
     - let l_2 be the number of components of J_j such that $\underline{J_{jk}} > 0$ or $\overline{J_{jk}} < 0$
     - if $l_2 > l_1$, then $(\underline{F_u}, \overline{F_u})$=`Evaluate`$F_j((\underline{x_1}, \overline{x_1}), \ldots, x_l, \ldots, x_m, \ldots, (\underline{x_m}, \overline{x_m}))$
     - otherwise $(\underline{F_u}, \overline{F_u}) = F_j((\underline{x_1}, \overline{x_1}), \ldots, x_l, \ldots, x_m, \ldots, (\underline{x_m}, \overline{x_m}))$
     - if this is the first estimation of F_j then $F_j = F_u$
     - otherwise
       * if $\underline{F_u} < \underline{F_j}$, then $\underline{F_j} = \underline{F_u}$
       * if $\overline{F_u} > \overline{F_j}$, then $\overline{F_j} = \overline{F_u}$
   - otherwise
     - $(\underline{F_u}, \overline{F_u}) = F_j((\underline{x_1}, \overline{x_1}), \ldots, x_l, \ldots, x_m, \ldots, (\underline{x_m}, \overline{x_m}))$
     - if this is the first estimation of F_j then $F_j = F_u$
     - otherwise
       * if $\underline{F_u} < \underline{F_j}$, then $\underline{F_j} = \underline{F_u}$
       * if $\overline{F_u} > \overline{F_j}$, then $\overline{F_j} = \overline{F_u}$
4. `end loop:`
5. otherwise $(\underline{F_j}, \overline{F_j}) = F_j((\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_m}, \overline{x_m}))$
6. return $(\underline{F_j}, \overline{F_j})$

This procedure has to be repeated for each F_j.

2.4.1.2 Improving the evaluation using the Jacobian and centered form

Let x_j^m be the middle point of $(\underline{x_j}, \overline{x_j})$ and $X = \{(\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_m}, \overline{x_m})\}$ be the box. Then:

$$(\underline{F_j(X)}, \overline{F_j(X)}) \in F_j(x_1^m, \ldots, x_m^m) + \sum_{i=1}^{i=m} ((\underline{x_i}, \overline{x_i}) - x_i^m) J_j((\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_i}, \overline{x_i}), x_{i+1}^m, \ldots, x_m^m) \tag{2.4}$$

see [5], pp. 52. This expression enable to get in some cases a sharper bound on F_j.

This evaluation is embedded into the evaluation procedure of the solving algorithms using the Jacobian. It is also available in its general form as

```
INTERVAL_VECTOR Centered_Form(int DimVar,int DimEq,
          INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
          VECTOR &Center,
          INTERVAL_VECTOR &Input)
```

where

- `DimVar`: number of variables
- `DimEq`: number of expressions
- `TheIntervalFunction`: procedure in `MakeF` format for interval evaluating the expressions
- `Gradient`: procedure un `MakeJ` format for evaluating the derivatives of the expressions
- `Center`:the center point for the centered form
- `Input`: the ranges for the variables

A variant of this procedure is

```
INTERVAL Centered_Form(int k,int DimVar,int DimEq,
          INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
          VECTOR &Center,
          INTERVAL_VECTOR &Input)
```

which is used to evaluate only expression number k.

A more sophisticated evaluation for the centered form is based on Baumann theorem [18]. First we define the procedure *cut(double x,INTERVAL X)* as:

$$cut(x,X) = \begin{cases} \overline{X}\ if\ x \geq \overline{X} \\ \underline{X}\ if\ x \leq \underline{X} \\ x\ otherwise \end{cases}$$

For a system of m equations F in n unknowns X we define

$$p_k^l = cut(\frac{Mid((\partial F_l'/\partial X_k)(X))}{Diam((\partial F_l'/\partial X_k)(X))}, [-1,1])$$

For a given equation l we use the centered form with as center

$$x_k^1 = Mid(X_k) - p_k^l Diam(X_k) \quad x_k^2 = Mid(X_k) - p_k^l Diam(X_k)$$

with k in $[1,n]$. The choice for x^1, x^2 is based on the property that the lower end-point of the centered form $F(x,X)$ has a maximum at x^2 while its upper end-point has a minimum at x^1. The interval evaluation of F_l is obtained as $F_l(x^1,X) \cap F_l(x^2,X)$. Although $2nm$ centered form are used to compute the interval evaluation of the m equations the calculation is in fact not so expensive as the interval evaluation of the Jacobian matrix has to be done only once.

The implementation is:

```
INTERVAL_VECTOR BiCentered_Form(int DimVar,
          int DimEq,
          INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
          INTERVAL_VECTOR &Input,
          int Exact)
```

where

- `DimVar`: number of variables
- `DimEq`: number of expressions
- `TheIntervalFunction`: procedure in `MakeF` format for interval evaluating the expressions
- `Gradient`: procedure un `MakeJ` format for evaluating the derivatives of the expressions
- `Input`: the ranges for the variables
- `Exact`: if 0 the procedure will return as soon as an interval evaluation of one expression does not include 0

A variant for evaluating only equation number k is

```
INTERVAL_VECTOR BiCentered_Form(int k,int DimVar,
          int DimEq,
          INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
          INTERVAL_VECTOR &Input,
          int Exact)
```

Another variant is based on the property that the numerical interval evaluation of the product `J(Input)(Input-Center)` may be overestimated as there may be several occurence of the same variable in this product. We may assume that this product has been computed symbolically, then re-arranged to reduce the number of occurence of the same variable leading to a procedure in `MakeF` format that computes directly the product. The syntax of the bicentered form procedure is

```
INTERVAL_VECTOR BiCentered_Form(int DimVar,
          int DimEq,
          INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* ProdGradient)(int, int, INTERVAL_VECTOR &),
          INTERVAL_VECTOR &Input,
          int Exact)
```

where `ProdGradient` is the procedure that computes the product `J(Input)(Input-Center)`, being understood that the `Center` is available in the global variable `ALIAS_Center_CenteredForm`.

2.4.1.3 Single bisection mode

We may use the single bisection mode i.e. bisect only one variable at a time. Fives modes exist for determining the variable to be bisected, the choice being made by setting `Single_Bisection` to a value from 1 to 8

- 1 : we just split the variable having the largest width
- 2 : this mode is based on the *smear function* as defined by Kearfott [7]: let $J = ((J_{ij}))$ be the Jacobian matrix of the system and let define for the variable x_i the smear value $s_i = \mathrm{Max}(|\underline{J_{ij}[\underline{x_i}, \overline{x_i}]}|, |\overline{J_{ij}[\underline{x_i}, \overline{x_i}]}|$ $\forall j \in [1, n]$ where n is the total number of functions. The variable that will be bisected will be the one having the largest s_i. There is however a drawback f the smear function: let consider for example the equation $F = f^2b^3 - 1 = 0$ where f, b are large identical intervals centered at 0. The derivative of F with respect to f is $2fb^3$ and with respect to b $3f^2b^2$: multiplied by the width of the interval we get $2f^2b^3$ and $3f^2b^3$. Hence the smear function for b will be in general larger than for f and b will always be bisected until its width is lower than the desired accuracy. Another example in which the smear function is not the best choice is presented in section 2.4.3.4. However the smear function is very often the most efficient mode and should be privileged.
- 3 : this is similar to the smear function except that we take into account its drawback. To avoid bisecting over and over the same variable we impose that a variable may be considered for bisection only if the ratio of its width over the maximal width of the box is not lower than the variable `ALIAS_Bound_Smear` (default value 1.e-5).

- 4 : this mode is based on the *Krawczyk operator*: to determine which variable should be bisected we consider the box $P = \{[\underline{x_1}, \overline{x_1}], \ldots, [\underline{x_n}, \overline{x_n}]\}$. When dealing with the variable x_i the single bisection mode will lead to two new boxes P_1, P_2. Let X_1, X_2 be the middle point of these boxes We have seen (section 2.10) that a fundamental point of Moore test for determining the unicity of a solution in a box is that $r_i(P_j) = ||I - J^{-1}(X_j)J(P_j)|| \leq 1$. Thus we will consider in turn each of the variable and compute the value of r_i for both P_1, P_2. The bisected variable will be chosen as the one leading to the minimal value of all r_i. However to avoid bisecting over and over the same variable we use another test: let d_i be the width of the interval $[\underline{x_i}, \overline{x_i}]$ and d_{max} be the maximum of all the d_i. If $d_i/d_{max} < 0.1$ we don't consider the variable x_i as a possible bisection direction.

- 5: we use a *round-robin* mode i.e. each variable is bisected in turn (first x_1, then x_2 and so on) unless the width of the range for the variable is less than the desired accuracy on the variable, in which case the bisected variable is the next one having a sufficient width

- 6: like mode 2 of `SolveGeneral`. `ALIAS_RANDG` may still be used to switch between mode 1 and mode 2 of `SolveGeneral`

- 7: like mode 2 of `SolveGeneral` except that it is assumed that the user has defined a simplification procedure that may allow to reduce the box directly within the bisection process

- 8: the variable are regrouped by groups of `ALIAS_Tranche_Bisection` elements. The bisection will look at each group in turn and bisect the first group that has elements whose diameter is larger than `ALIAS_Size_Tranche_Bisection`. When the element of the group have all elements whose diameter is lower than this threshold the bisection will consider the next group. If all elements of all groups have a diameter lower than the threshold the smear function will be used to determine which variable will be bisected.

The smear mode leads in general to better result than the other modes (but there are exception, see example in section 2.4.3.4). There is another mode called the *mixed bisection*: among the n variables we will bisect $m_1 < n$ variables, which will lead to 2^{m_1} new boxes. This mode is obtained by setting the global integer variable `ALIAS_Mixed_Bisection` to $m_1 - 1$. Here we will order the variables according to the value of their smear function (if the flag `Single_Bisection` is 2 or 3) or according to their width (for 1,4,5).

2.4.2 Implementation

The algorithm is implemented in generic form as:

```
int Solve_General_Gradient_Interval(int m,int n,
  INTEGER_VECTOR Type_Eq,
          INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* IntervalGradient)(int, int, INTERVAL_VECTOR &),
          INTERVAL_VECTOR & TheDomain,
          int Order,int M,int Stop,
          double epsilon,double epsilonf,double Dist,
          INTERVAL_MATRIX & Solution,int Nb,INTEGER_MATRIX &GI,
          int (* Simp_Proc)(INTERVAL_VECTOR &))
```

the arguments being:

- `m`: number of unknowns

- `n`: number of functions, see the note 2.3.4.1

- `Type_Eq`: type of the functions, see the note 2.3.4.2

- `IntervalFunction`: a function which return the interval vector evaluation of the functions, see the note 2.3.4.3. Remember that if you have equations and inequalities in the system you must first define the equations and then the inequalities.

- `IntervalGradient`: a function which the interval matrix of the jacobian of the functions, see the note 2.4.2.2
- `TheDomain`: box in which we are looking for solution of the system. A copy of the search domain is available in the global variable `ALIAS_Init_Domain`
- `Order`: the type of order which is used to store the boxes created during the bisection process. This order may be either `MAX_FUNCTION_ORDER` or `MAX_MIDDLE_FUNCTION_ORDER`. See the note on the order 2.3.4.4.
- `M`: the maximum number of boxes which may be stored. See the note 2.3.4.5
- `Stop`: the possible values are 0,1,2
  - 0: the algorithm will look for every solution in `TheDomain`
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found
- `epsilon`: the maximal width of the box, see the note 2.3.4.6
- `epsilonf`: the maximal width of the function intervals, see the note 2.3.4.6. Note that if this value is set to 0, then Moore test is not used.
- `Dist`: minimal distance between the middle point of two interval solutions, see the note 2.3.4.7
- `Solution`: an interval matrix of size (`Nb`,`m`) which will contained the solution intervals. This list is sorted using the order specified by `Order`
- `Nb`: the maximal number of solution which will be returned by the algorithm
- `GI`: an integer matrix called the *simplified jacobian*, which give a-priori information on the sign of the derivative of the function. `GI(i,j)` indicates the sign of the derivative of function `i` with respect to variable `j` using the following code:
  - -1: the derivative is always negative
  - 0: the function is not dependent of variable `j`
  - 1: the derivative is always positive
  - 2: the sign of the derivative is not known
- `Simp_Proc`: a user-supplied procedure that take as input the current box and proceed to some further reduction of the width of the box components or even determine that there is no solution for this box, in which case it should return -1 (see note 2.3.3). Remember that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2).

Note that the following arguments may be omitted:

- `Type_Eq`: in that case all the functions will supposed to be equations.
- `GI`: in that case all the sign of the derivatives will supposed to be unknown
- `Simp_Proc`: no simplification procedure is provided by the user

2.4.2.1 Return code

The procedure will return an integer k

- $k \geq 0$: number of solutions
- $k = -1$: the size of the storage is too low (possible solutions: increase `M`, or use the 3B method, or use the reverse storage mode or the single bisection mode)
- $k = -2$: `m` or `n` is not strictly positive

- $k = -3$: `Order` is not 0 or 1
- $k = -4$: one of the function in the system has not a type 0, -1 or 1 (i.e. it's not an equation, an inequality $F \leq 0$ or an inequality $F \geq 0$)
- $k = -5$: we are in the optimization mode and more than one functions are expressions to be optimized (see the Optimization chapter)
- $k = -100$: in the mixed bisection mode the number of variables that will be bisected is larger than the number of unknowns
- $k = -200$: one of the value of `ALIAS_Delta3B` or `ALIAS_Max3B` is negative or 0
- $k = -300$: one of the value of `ALIAS_SubEq3B` is not 0 or 1
- $k = -400$: although `ALIAS_SubEq3B` has as size the number of equations none of its components is 1
- $k = -500$: `ALIAS_ND` is different from 0 (i.e. we are dealing with a non-0 dimensional problem, see the corresponding chapter) and the name of the result file has not been specified
- $k = -1000$: the value of the flag `Single_Bisection` is not correct
- $k = -3000$: we use the full bisection mode and the problem has more than 10 unknowns

The following variables play also a role in the computation:

- `ALIAS_Store_Gradient`: if not 0 the gradient matrix of each box will be stored together with the input intervals. Must be set to 0 for large problem (default value: 1)
- `ALIAS_Diam_Max_Gradient`: if the maximal width of the ranges in a box is lower than this value, then the gradient will be used to perform the interval evaluation of the functions (default value: 1.e10)
- `ALIAS_Diam_Max_Kraw`: if the maximal width of the ranges in a box is lower than this value, then the Krawczyk operator will be used to determine if there is a unique solution in the box (default value: 1.e10)
- `ALIAS_Diam_Max_Newton`: if the maximal width of the ranges in a box is lower than this value, then the interval Newton method will be used either to try to reduce the width of the box or to to ensure that there is no solution of the system in the box (default value: 1.e10)

2.4.2.2 Jacobian matrix

The user must provide a function which will compute the interval evaluation of the jacobian matrix of its particular functions for a given box. As for the function evaluation procedure we have chosen a syntax which shows the best compromise between program calls and interval calculation. The syntax of this function is:

```
INTERVAL_MATRIX IntervalGradient (int l1,int l2,INTERVAL_VECTOR & x)
```

- `x`: a m dimensional interval vector which define the intervals for the unknowns
- `l1,l2`: the function must be able to return the interval evaluation of the component of the jacobian matrix at row `l1` and column `l2` i.e. the derivative of the function number `l1` with respect to the variable number `l2`. The first row has number 1, the last `n` and the first column has number 1, the last `m`.

This procedure returns an interval matrix `grad` of size `m X n` in which `grad(l1,l2)` has been computed. This function should be written using the BIAS/Profil rules.

Note that if `ALIAS_Store_Gradient` has not been set to 0 we will store the simplified Jacobian matrix for each box. Indeed if for a given box B the interval evaluation of one element of the gradient has a constant sign (indicating a monotonic behavior of the function) setting the simplified jacobian matrix element to -1 or 1 allows to avoid unnecessary evaluation of the element of the Jacobian for the box resulting from a bisection of B as they will exhibit the same monotonic behavior. Although this idea may sound quite simple it has a very positive effect on the computation time. The name of the storage variable of the simplified jacobian is:

```
INTEGER_MATRIX Gradient_Solve_General_Interval
```

If a function `i` is not differentiable you just set the value of `grad(i,..)` to the interval [-1e30,1e30]. It is important to define here a large interval as the program `Compute_Interval_Function_Gradient` that computes the interval evaluation of the function by using their derivatives uses also the Taylor evaluation based on the value of the derivatives: a small interval value of the derivatives may lead to a wrong function evaluation.

Note also that a convenient way to write the `Gradient` procedure is to use the procedure `MakeJ` offered by ALIAS-Maple (see the ALIAS-Maple manual). Take care also of the interval valuation problems of the element of the Jacobian (see section 2.1.1.3) which may be different from the one of the functions: for example if a function is $\sqrt{x}F(x, y, \ldots)$ it is interval-valuable as soon as the lower bound of x is greater or equal to 0 although the gradient will involve $1/\sqrt{x}$ which is not interval-valuable if the lower bound of x is equal to 0. ALIAS-Maple offers also the possibility to treat automatically the interval-valuation problems of the jacobian elements.

2.4.2.3 Evaluation procedure using the Jacobian

A better evaluation of the function intervals than the `IntervalFunction` can be obtained using the Jacobian matrix. A specific procedure can be used to obtain this evaluation:

```
INTERVAL_VECTOR Compute_Interval_Function_Gradient(int m,int n,
        INTEGER_VECTOR &Type_Eq,
        INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
        INTERVAL_MATRIX (* IntervalGradient)(int, int, INTERVAL_VECTOR &),
        INTERVAL_VECTOR & Input, int Exact,
        INTEGER_VECTOR &AG,INTEGER_MATRIX &AR)
```

This procedure computes the function intervals for the box `Input`.

- `Type_Eq` is an integer array whose `Dimension_Eq` elements indicates the nature of the functions: -1 for inequality <= 0, 0 for equation, 1 for inequality >= 0, -2 for a function to be minimized, 2 for a function to be maximized and 10 for a function to be both minimized and maximized (note that for an optimization problem the function that has to be minimized must be the last function in the list of function).
- the integer `Exact` should be put to 1 as for a value of 0 the procedure stop the evaluation of each box as soon as the lower bound of the interval is negative and the upper bound positive.
- `AG` is an integer vector of size `m x n` which indicates if the sign of some derivatives are already known (the elements should then have the values -1, 0 or 1) or not (the value must then be 2)
- `AR` is a return matrix with the sign of the derivatives for `Input`

The parameters `Type_Eq`, `AG, AR` may be omitted. This procedure uses the derivatives for improving the interval evaluation of the functions in two different ways:

- by taking into account of the monotonicity of the functions
- by using an interval evaluation of the functions based on their Taylor expansion: it is therefore necessary to evaluate rightly the derivatives of the functions

The best evaluation of the l-th equation may be computed with

```
INTERVAL Compute_Interval_Function_Gradient_Line(int l,int Dim_Var,
int Dimension_Eq,
INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
INTERVAL_VECTOR &Input,int Exact,INTEGER_VECTOR &AG)
```

2.4.2.4 Storage

The boxes generated by the bisection process are stored in an interval matrix:

`Box_Solve_General_Interval(M,m)`

while the corresponding simplified Jacobian matrix is stored in the integer matrix of size (M, m $\times$ n):

`Gradient_Solve_General_Interval`

called the *simplified jacobian*: the entry i, j of this matrix indicates for function i and variable j that the gradient is always positive (1), the gradient is always negative (-1), the function does not depend upon the variable (0), the gradient may have not a constant sign within the range of the variable (2). The purpose of storing the simplified gradient for each box is to avoid to re-compute a gradient as soon as it has been determined that a father of the box has already a gradient with a constant sign. This has the drawback that for large problems this storage will be also large: hence it is possible to avoid this storage by setting the variable `ALIAS_Store_Gradient` to 0 (its default value is 1).

The algorithm try to manage the storage in order to solve the problem with the given number M. As seen in section 2.3.1.2 two storage modes are available, the *Direct Storage* and the *Reverse Storage* modes, which are obtained by setting the global variable `Reverse_Storage` to 0 (the default value) or 1.

For both modes the algorithm will first run until the bisection of the current box leads to a total number of boxes which exceed the allowed total number. It will then delete the boxes in the list which have been already bisected, thereby freeing some storage space (usually larger for the reverse mode than for the direct mode) and will start again.

2.4.3 Examples

2.4.3.1 Example 1

This problem has been presented in section 2.3.5.1.

The `IntervalGradient` function is defined as:

```
INTERVAL_MATRIX IntervalGradient (int l1,int l2,INTERVAL_VECTOR & x)
{
INTERVAL_MATRIX Grad(3,3);
if(l1==1)
      {
      if(l2==1){Grad(1,1)=2*x(1)+2;return Grad;}
      if(l2==2){Grad(1,2)=0;return Grad;}
      if(l2==3){Grad(1,3)=0;return Grad;}
      }
if(l1==2)
      {
      if(l2==1){Grad(2,1)=0;return Grad;}
      if(l2==2){Grad(2,2)=2*x(2)+2;return Grad;}
      if(l2==3){Grad(2,3)=0;return Grad;}
      }
if(l1==3)
      {
      if(l2==1){Grad(3,1)=0;return Grad;}
      if(l2==2){Grad(3,2)=0;return Grad;}
      if(l2==3){Grad(3,3)=2*x(3)+2;return Grad;}
      }
}
```

A test main program may now be written as:

```
INT main()
{
int Num,i,j,order,precision,Stop;
// accuracy of the solution either on the function  or on the variable
double Accuracy,Accuracy_Variable,Diff_Sol;
INTERVAL_MATRIX SolutionList(200,3);//the list of solutions
INTERVAL_VECTOR TestDomain;//the input intervals for the variable
INTERVAL_VECTOR F(3);

//We set the value of the variable intervals
SetTestDomain (TestDomain);

cerr << "Accuracy on Function = "; cin >> Accuracy;
cerr << "Accuracy on Variable = "; cin >> Accuracy_Variable;
```

```
cerr << "Order (0,1)"; cin >>order;
cerr << "Stop at first solutions (0,1,2):";cin>>Stop_First_Sol;
cerr << "Separation between distincts solutions:";cin>> Diff_Sol;

//let's solve....
Num=Solve_General_Gradient_Interval(3,3,IntervalTestFunction,
        IntervalGradient,TestDomain,order,10000,Stop,
      Accuracy_Variable,Accuracy,Diff_Sol,SolutionList,1);

//too much intervals have been created, this is a failure
if(Num== -1)cout<<"Procedure has failed (too many iterations)"<<endl;

//otherwise print the solution intervals
for(i=1;i<=Num;i++)
  {
    cout << "solution " << i <<endl;
    cout << "x(1)=" << SolutionList(i,1) << endl;
    cout << "x(2)=" << SolutionList(i,2) << endl;
    cout << "x(3)=" << SolutionList(i,3) << endl;
    cout << "Function value at this point" <<endl;
    for(j=1;j<=3;j++)F(j)=SolutionList(i,j);
    cout << Compute_Interval_Function_Gradient(Dimension,Dimension_Eq,
     IntervalTestFunction,
     IntervalGradient,
     F,1) << endl;
  }
  return 0;
}
```

A property of this problem is that the Jacobian of the system is singular at the solution. Hence the unicity test cannot be verified as it needs to evaluate the inverse of the jacobian matrix (as will fail the classical Newton scheme, see section 2.9, that needs also the inverse Jacobian). But even with `epsilon`=`epsilonf`=1e-6 the algorithm is able to find an approximation of the solution with 16 boxes only. Interestingly this is a case where the 3B method is not efficient at all: with the 3B method the number of boxes increases to over 100 000. This is quite normal: as the 3B method is used before the interval Newton method it reduces the range for the unknowns toward a region where the interval Newton method will fail as the Jacobian is close to a singularity. We therefore end up with a solving that is only based on the bisection process and we have seen that this process behaves poorly for this system. But if we mix the 3B method and the 2B filtering of section 2.17 then the solving needs only 1 box.

2.4.3.2 Example 2

The problem we want to solve has been presented in section 2.3.5.2,15.1.1.

The `IntervalGradient` procedure is:

```
INTERVAL_MATRIX IntervalGradient (int l1,int l2,INTERVAL_VECTOR & in)
{
INTERVAL_MATRIX Grad(3,3);
INTERVAL x,y,teta;
x=in(1);y=in(2);teta=in(3);
if(l1==1)
  {
    if(l2==1){Grad(1,1)=2*x;return Grad;}
    if(l2==2){Grad(1,2)=2*y;return Grad;}
    if(l2==3){Grad(1,3)=0;return Grad;}
  }
if(l1==2)
  {
    if(l2==1){Grad(2,1)=2.0*x-20.0+8.0*Cos(teta);return Grad;}
    if(l2==2){Grad(2,2)=2.0*y+8.0*Sin(teta);return Grad;}
    if(l2==3){Grad(2,3)=-8.0*x*Sin(teta)+80.0*Sin(teta)+8.0*y*Cos(teta);
      return Grad;}
  }
if(l1==3)
  {
    if(l2==1){Grad(3,1)=2.0*x-6.0+4.0*Cos(teta)-4.0*Sin(teta);
      return Grad;}
    if(l2==2){Grad(3,2)=2.0*y-20.0+4.0*Sin(teta)+4.0*Cos(teta);return Grad;}
    if(l2==3){
      Grad(3,3)=52.0*Sin(teta)-28.0*Cos(teta)+(4.0*Cos(teta)-
              4.0*Sin(teta))*y+(-4.0*Sin(teta)-4.0*Cos(teta))*x;
      return Grad;
             }
  }
}
```

We may use the same main program as in the previous example (the name of this program is `Test_Solve_General1_Gradient`).

Let's assume that we set `epsilonf` to 0 and `epsilon` to 0.01 while looking at all the solutions (`Stop`=0), using the maximum equation ordering and setting `Dist` to 0.1. The algorithm provide the following solutions after using 55 boxes:

$$x = [3.36607, 3.37306] \quad y = [6.21468, 6.21856] \quad \theta = [-0.808416, -0.805665]$$
$$x = [4.99845, 5.00165] \quad y = [4.99845, 5.00146] \quad \theta = [-0.000536641, 0]$$

We notice that all of the roots are contained in the solution intervals.

If we use the maximum middle-point equation ordering the algorithm provide the same solution intervals.

With `epsilonf`=0.001, `epsilon`=0 the algorithm still find exactly the root with 55 boxes and a computation time of 7010ms. Here Moore test may have failed as the solution in θ is 0, which correspond exactly to split point in the bisection process: it may be useful to break the symmetry in the test domain.

Using the single bisection mode and setting the flag `Single_Bisection` to 2 enable to reduce the number of boxes to 33 and the computation time to 3580ms for `epsilonf`=0.00001.

Note that we may improve the efficiency of the procedure by using simplification procedures such as the 2B (section 2.17) and the 3B method. In that case for `epsilonf`=1e-6, `epsilon`=1e-6 the number of boxes will have been reduced to 7. Note that the solution [5,5,0] is still not guaranteed. But using a search space of $[-\pi, \pi + 1]$ for θ allow the Moore test to guarantee both solutions.

2.4.3.3 Example 3

This example is derived from example 2. We notice that in the three equations of example 2 the second degree terms of x, y are for all functions $x^2 + y^2$. Thus by subtracting the first function to the second and third we get a linear system in x, y. This system is solved and the value of x, y are substituted in the first function. We get thus a system of one equation in the unknown θ (see section 15.1.2). The roots of this equation are 0,-0.8067834380. The test program is `Test_Solve_Gradient_General2`. The `IntervalGradient` function is written as:

```
INTERVAL_MATRIX IntervalGradient (int l1,int l2,INTERVAL_VECTOR & in)
{
INTERVAL_MATRIX Grad(1,1);
Grad(1,1)=-3788.0+(2692.0+(7576.0-4800.0*Cos(in(1)))*Cos(in(1)))*Cos(in(1))+(
25912.0+(-39320.0+14520.0*Cos(in(1)))*Cos(in(1)))*Sin(in(1));
return Grad;
}
```

With `epsilonf`=0 and `epsilon`=0.001 we get the solution by using 8 boxes:

$$\theta = [-0.0005316396, 0] \qquad \theta = [-0.8071015, -0.8063247]$$

for whatever order. The solution intervals contain the roots of the equation. If we use `epsilon`=0 and `epsilonf`=0.1 we get by using 8 boxes:

$$\theta = [-0.8067839277] \quad \theta = [-5.4136579e - 16]$$

Here we get a unique solution and a range solution. But we notice that the solution 0 is exactly the middle point of the test domain: Moore test will fail as 0 will always be an end-point of the range. If we break the symmetry of the test domain we will get exactly both solutions.

2.4.3.4 Example 4

In this example (see section 15.1.3) we deal with a complex problem of three equations in three unknowns ψ, θ, ϕ. We are looking for a solution in the domain:

$$[4.537856054, 4.886921908], [1.570796327, 1.745329252], [0.6981317008, 0.8726646262]$$

The system has a solution which is approximatively:

$$4.6616603883, 1.70089818026, 0.86938888189$$

This problem is extremely ill conditioned as for the TestDomain the functions intervals are:

$$[-1.45096e+08, 1.32527e+08]; [-38293.3, 29151.5]; [-36389.1, 27705.7]$$

The name of the test program is Test_Solve_General_Gradient. With espsilonf=0 and epsilon=0.001 and if we stop at the first solution we find with the maximum equation ordering:

$$\psi = [4.661209, 4.661906] \quad \theta = [1.700332, 1.701014] \quad \phi = [0.8693552, 0.8696447]$$

with 73 boxes using the direct storage mode (with the reverse storage mode only 37 boxes are needed). With the maximum middle-point equation ordering we find the same intervals with 67 boxes in the direct storage mode (41 for the reverse storage mode).

The importance of normalizing the functions has been mentioned in section 2.3.5.4. But in this example the use of the Jacobian matrix enable to drastically reduce the computation time. If we use epsilonf=0.1 and epsilon=0 and if we stop at the first solution we find an exact solution using 73 boxes:

$$\psi = [4.66166] \quad \theta = [1.700898] \quad \phi = [0.869388]$$

in a time which is about 1/100 of the time necessary when we don't use the Jacobian.

In the single bisection smear mode (i.e. only one variable is bisected in the process) the same root is obtained in 21080ms (about 50% less time than when using the full bisection mode) with only 40 boxes in the direct storage mode.

Note that for epsilonf=0.1 and epsilon=0 we find the only root with 73 boxes in 39760ms (41 boxes and 24650ms in the single bisection smear mode).

Note that we may improve the efficiency of the procedure by using simplification procedures such as the 2B (section 2.17) and the 3B method. An interesting point in this example is that the bisection mode 1 (bisecting along the variable whose interval has the largest diameter) is more effective than using the bisection mode 2 (using the smear function) with 53 boxes against 108 for the mode 2 for epsilonf=1e-6 and epsilon=1e-6. This can easily been explained by the complexity of the Jacobian matrix elements that leads to a large overestimation of their values when using interval: in that case the smear function is not very efficient to determine which variable has the most influence on the equations.

2.4.4 General comments

According to the system this procedure may not be especially faster than the general purpose algorithm but the number of necessary boxes is in general drastically reduced. Furthermore the use of Moore test and interval Newton method enable in many cases to determine exactly the solutions.

This solving algorithm may be tested on-line on the web page ALIAS-on-line that can be found on the COPRIN web page.

2.5 General purpose solving algorithm with Jacobian and Hessian

2.5.1 Mathematical background

In this new algorithm we will try to improve the evaluation of the function intervals by using the Hessian of the functions. This improvement is based on a sharper analysis of the monotonicity of the functions which in turn is based on a sharper evaluation of the Jacobian matrix of the system. The element J_{ij} of the Jacobian matrix at row i and column j is:

$$\frac{\partial F_i}{\partial x_j}$$

Now consider the corresponding line of the Hessian matrix which is:

$$H_{ij} = \frac{\partial F_i}{\partial x_j \partial x_k} \text{ with } k \in [1, m]$$

The elements of this line indicate the monotonic behavior of the elements of the Jacobian matrix. If some elements in this line have a constant sign, then the elements of the Jacobian are monotonic with respect to some of the unknowns and using this monotonicity we may improve the evaluation of the element of the Jacobian matrix. This improvement has to be applied recursively: indeed as we will evaluate the Jacobian elements for boxes in which some components have now a fixed value the evaluation of the Hessian matrix for these boxes may lead to a larger number of the component of the Hessian which have a constant sign. The recursion will stop if all the component of the Hessian line have a constant sign or if the number of component with a constant sign does not increase.

Note also that not all the function must be differentiable to use this procedure: only one of them is sufficient. In that case you will have however to set special values in the gradient and hessian function (see 2.4.2.2).

We will also use the Hessian in order to try to sharpen the evaluation of J_{ij}. Let the box intervals be $\{(\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_n}, \overline{x_n})\}$. Let x_j^m be the middle point of $(\underline{x_j}, \overline{x_j})$ and $X = \{(\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_m}, \overline{x_m})\}$ be the box. Then:

$$(\underline{J_{ij}(X)}, \overline{J_{ij}(X)}) \in J_{ij}(x_1^m, \ldots, x_m^m) + \sum_{i=1}^{i=m}((\underline{x_i}, \overline{x_i}) - x_i^m)H_{ij}((\underline{x_1}, \overline{x_1}), \ldots, (\underline{x_i}, \overline{x_i}), x_{i+1}^m, \ldots, x_m^m) \quad (2.5)$$

see [5], pp. 52. This expression enable to get in some cases a sharper bound on J_{ij}.

The improvement of the evaluation of the function intervals is due to the fact that a sharper evaluation of the Jacobian matrix may lead to a larger number of Jacobian elements with constant sign than with the direct evaluation of the Jacobian matrix. To speed up the algorithm we store the Jacobian matrix of each box: this enable to avoid the evaluation of the components of the Jacobian matrix which have a constant sign when bisecting the box

Another interest of the Hessian is that it enable to use Kantorovitch theorem. This theorem (see section 3.1.2) can be applied if the number of unknowns is equal to the number of equations and enable to determine boxes in which there is an unique solution, solution which can be found using Newton method (see section 2.9) with as estimate of the solution any point within the boxes.

We will use this theorem at three possible levels:

- level 0: we want solution intervals for which the maximal width is equal or lower than a given threshold. In that case imagine that two solution intervals have been found at some point of the algorithm, this two solutions being close. We will apply Kantorovitch theorem for the center of the two solution intervals. In the most favorable case one of them will contain an unique solution while the boxes given by Kantorovitch theorem will cover the other one: consequently this input intervals will be eliminated of the solution intervals. Therefore Kantorovitch theorem will eliminate spurious solution intervals and we don't need to indicate a minimal distance between the solution intervals.

- level 1: we look for solution intervals whose width has no importance as soon as we are sure that they contain one unique solution which can be found using Newton method with as estimate of the solution any point within the solution intervals. In that case we will apply Kantorovitch theorem for each boxes which appear during the algorithm. If the theorem give a solution intervals we will store it in the solution list and update the remaining boxes of the list so that none of them contain the solution intervals. A consequence is that the width of the solution intervals cannot be determined beforehand while each solution intervals that have been determined using this method contain one unique solution which can be determined using Newton method.

- level 2: we apply Newton method for each box and if Newton converge toward a solution within the current box we store the box as solution interval. The boxes in the list are then filtered so that none of them contains the solution interval.

Furthermore we use the inflation method presented in section 3.1.6 to increase the width of the box in which we may guarantee that there is a unique solution.

As for the method using only the gradient we use the Hansen-Sengupta version of the interval Newton method to improve the boxes (see [21]).

2.5.1.1 Single bisection mode

Instead of bisecting all the variables we may bisect only one of the variable. The criteria for determining which of the variable will be bisected is identical to the one presented in section 2.4.1.3 for the mode up to 5. The mode 6 is different: basically we will bisect the variable having the largest diameter except that it is supposed that a weight is assigned to each variable and the diameter used by the bisection process is the diameter of the range for the variable multiplied by the weight of the variable. The weight must be indicated in the vector `ALIAS_Bisection_Weight`.

2.5.2 Implementation

The generic implementation of this solving procedure is:

```
int Solve_General_JH_Interval(int m,int n,
       INTEGER_VECTOR Type_Eq,
       INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* IntervalHessian)(int,int,INTERVAL_VECTOR &),
       INTERVAL_VECTOR & TheDomain,
       int Order,int M,int Stop,
       double epsilon,double epsilonf,
       INTERVAL_MATRIX & Solution,INTEGER_VECTOR & IsKanto,
       int ApplyKanto,int Nb,
       INTERVAL_MATRIX &GM,
       int (* Simp_Proc)(INTERVAL_VECTOR &))
```

the arguments being:

- `m`: number of unknowns
- `n`: number of functions, see the note 2.3.4.1
- `Type_Eq`: type of the functions, see the note 2.3.4.2
- `IntervalFunction`: a function which return the interval vector evaluation of the functions, see the note 2.3.4.3. Remember that if you have equations and inequalities in the system you must first define the equations and then the inequalities.
- `IntervalGradient`: a function which return the interval matrix of the jacobian of the functions, see the note 2.4.2.2
- `IntervalHessian`: a function which return the interval matrix of the Hessian of the functions, see the note 2.5.2.1
- `TheDomain`: box in which we are looking for solution of the system. A copy of the search domain is available in the global variable `ALIAS_Init_Domain`
- `Order`: the type of order which is used to store the intervals created during the bisection process. This order may be either `MAX_FUNCTION_ORDER` or `MAX_MIDDLE_FUNCTION_ORDER`. See the note on the order 2.3.4.4.
- `M`: the maximum number of boxes which may be stored. See the note 2.5.2.2
- `Stop`: the possible values are 0,1,2
  - 0: the algorithm will look for every solution in `TheDomain`
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found
- `epsilon`: the maximal width of the box, see the note 2.3.4.6
- `epsilonf`: the maximal width of the function intervals, see the note 2.3.4.6
- `Solution`: an interval matrix of size (`Nb`,`m`) which will contained the solution intervals. Each solution may be:

  - a set of intervals with the associated flag `IsKanto` to 0:
  - a set of intervals with the associated flag `IsKanto` to 1: there is an unique solution in the set and Newton method will converge toward this solution
  - a set of intervals reduced to a point with the associated flag `IsKanto` to 0: this point is a solution which has been obtained with Krawczyk method (see 2.10). The accuracy of this solution may be improved by using the point as starting point for Krawczyk method and decreasing the accuracy `epsilonf`

- `IsKanto`: an integer vector of dimension `Nb`. A value of 1 for `IsKanto(i)` indicate that applying Newton method (see section 2.9) with as estimate the center of the solution intervals `Solution(i)` will converge toward the unique solution which lie within the solution intervals `Solution(i)`
- `ApplyKanto`: an integer which indicate at which level we use Kantorovitch theorem. If 1 we use Kantorovitch theorem (see section 3.1.2 and the mathematical background) to determine the solution. A consequence is that the solution interval may have a width larger than `epsilon`. If 0 we use Kantorovitch theorem just to separate the solutions: the solution interval will have a width `epsilon`. If 2 we will apply Newton method for every box which has not been eliminated during the bisection process but we will consider the result a solution only if it lie within the box. The maximal number of iteration is determined by the global variable `Max_Iter_Newton_JH_Interval` (by default 100). In that case we may miss solutions if they are lying inside the same box.
- `Nb`: the maximal number of solution which will be returned by the algorithm
- `GM`: an interval matrix which give a-priori information on the values of the derivatives of the function. `GM(i,j)` is the interval value of the derivative of function `i` with respect to variable `j`
- `Simp_Proc`: a user-supplied procedure that take as input the current box and proceed to some further reduction of the width of the box or even determine that there is no solution for this box, in which case it should return -1 (see note 2.3.3). Remember that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2).

Note that the following arguments may be omitted:

- `Type_Eq`: in that case all the functions will supposed to be equations.
- `GM`: in that case all the derivatives will supposed to be unknown
- `Simp_Proc`: no simplification procedure is provided by the user

The following variables play also a role in the computation:

- `ALIAS_Store_Gradient`: if not 0 the gradient matrix of each box will be stored together with the boxes. Must be set to 0 for large problem (default value: 1)
- `ALIAS_Diam_Max_Gradient`: if the maximal width of the ranges in a box is lower than this value, then the gradient will be used to perform the interval evaluation of the functions (default value: 1.e10)
- `ALIAS_Diam_Max_Kraw`: if the maximal width of the ranges in a box is lower than this value, then the Krawczyk operator will be used to determine if there is a unique solution in the box (default value: 1.e10)
- `ALIAS_Diam_Max_Newton`: if the maximal width of the ranges in a box is lower than this value, then the interval Newton method will be used either to try to reduce the width of the box or to to ensure that there is no solution of the system in the box (default value: 1.e10)
- `ALIAS_Always_Use_Inflation`: if `ApplyKanto` is set to 1 we get for each solution a box B which contains only one solution. If this flag is set to 1 we compute the solution using Newton and then we use an inflation procedure that try to determine a box B_1 which is larger than B and contains also only one solution
- `ALIAS_Eps_Inflation`: the inflation algorithm will try to increase the width of the box B by at least this value

- `ALIAS_Sing_Determinant`: if the determinant of the jacobian matrix of the system is lower than this value, then the system is supposed to be singular

- `ALIAS_Diam_Sing`: for a value s of this parameter if there is singular solution in the system, then the algorithm will not look for solution of the system in a box of width $2s$ around the singular solution (default value: 0)

- `ALIAS_Use_Grad_Equation`: if this integer array has a size of `n` the derivatives of equation i will be used to evaluate the i-th equation only if `ALIAS_Use_Grad_Equation[i]` is not 0

- `ALIAS_No_Hessian_Evaluation`: if set to 0 we will not use the Hessian to sharpen the interval evaluation of the Gradient when performing the interval evaluation of the equations. This may be useful if its known that the interval evaluation of the elements of the hessian will have always a constant sign

2.5.2.1 Hessian procedure

The syntax of this function is:

```
Hess=INTERVAL_MATRIX IntervalHessian (int l1,int l2,INTERVAL_VECTOR & in)
```

This procedure should return an interval matrix of size `m` $\times$ `n`, `m` in which the Hessian of function numbered `l1` to `l2` has been updated (function number start at 1). The Hessian matrix of function i (which is of size `n` $\times$ `m`) is stored at location `Hess`((i-1)`m`+1...i`m`,1...`m`). Remember that for each function the Hessian matrix is symmetric: this fact should be used in order to speed up the evaluation of this matrix. If a function in the system is not C^2 you set all the elements of its hessian matrix to the interval [-1e30,1e30]. Remember also here to verify that each element of the Hessian should be interval-valuable (see section 2.1.1.3).

2.5.2.2 Storage

The boxes generated by the bisection process are stored in an interval matrix:

```
Box_Solve_General_Interval(M,m)
```

while the corresponding Jacobian matrix is stored in the interval matrix of size (`M`, `m` $\times$ `n`):

```
Gradient_Solve_JH_Interval
```

The purpose of storing the gradient for each box is to avoid to re-compute a gradient as soon as it has been determined that a father of the box has already a gradient with a constant sign. This has the drawback that for large problems this storage will be also large: hence it is possible to avoid this storage by setting the variable `ALIAS_Store_Gradient` to 0 (its default value is 1). Note that here we store the *interval* gradient matrix and not the *simplified* gradient matrix as in the solving procedure involving only the Jacobian.

The algorithm try to manage the storage in order to solve the problem with the given number `M` (see section 2.3.1.2). As seen in section 2.3.1.2 two storage modes are available, the *Direct Storage* and the *Reverse Storage* modes, which are obtained by setting the global variable `Reverse_Storage` to 0 (the default value) or to the number of unknowns+1.

For both modes the algorithm will first run until the bisection of the current box leads to a total number of boxes which exceed the allowed total number. It will then delete the boxes in the list which have been already bisected, thereby freeing some storage space (usually larger for the reverse mode than for the direct mode) and will start again.

If the procedure has to be used more than once it is possible to speed up the computation by allocating the storage space before calling the procedure. Then you may indicate that the storage space has been allocated beforehand by indicating a negative value for `M`, the number of boxes being given by the absolute value of `M`.

2.5.2.3 Improvement of the function evaluation and of the Jacobian

An improved value of the Jacobian is obtained by taking account its derivative in the procedure:

```
INTERVAL_MATRIX Compute_Best_Gradient_Interval(int Dimension,
                    int Dimension_Eq,
                    INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                    INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),
                    INTERVAL_VECTOR &Input,
                    int Exact,INTERVAL_MATRIX &InGrad)
```

where

- `Exact`: if 1 the calculation for one element of the Jacobian will stop as soon as the method has found that the interval evaluation of the element will not have a constant sign. If 0 the best interval evaluation will be computed
- `InGrad`: if this matrix is not the zero matrix we will assume that the non zero elements of this matrix are the interval evaluation of the Jacobian

To compute only the best value of the jacobian element at l-th row nad j-th column you may use:

```
INTERVAL Compute_Best_Gradient_Interval_line(int l,int j,int Dim,
                    int Dimension_Eq,
                    INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                    INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),
                    INTERVAL_VECTOR &Input,int Exact)
```

We may also obtain the best interval evaluation of the equations through the procedure

```
INTERVAL_VECTOR Compute_Interval_Function_Gradient(int Dimension,
                    int Dimension_Eq,
                    INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
                    INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                    INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),
                    INTERVAL_VECTOR &Input, int Exact)
```

2.5.2.4 Return code and debug

The procedure will return an integer k

- $k \geq 0$: number of solutions
- $k = -1$: the size of the storage is too low (possible solutions: increase `M`, or use the 3B method, or use the reverse storage mode or the single bisection mode)
- $k = -2$: `m` or `n` is not strictly positive
- $k = -3$: `Order` is not 0 or 1
- $k = -4$: one of the function in the system has not a type 0, -1 or 1 (i.e. it's not an equation, neither inequality $F \leq 0$ or an inequality $F \geq 0$)
- $k = -100$: in the mixed bisection mode the number of variables that will be bisected is larger than the number of unknowns
- $k = -200$: one of the value of `ALIAS_Delta3B` or `ALIAS_Max3B` is negative or 0
- $k = -300$: one of the value of `ALIAS_SubEq3B` is not 0 or 1
- $k = -400$: although `ALIAS_SubEq3B` has as size the number of equations none of its components is 1

- $k = -500$: `ALIAS_ND` is different from 0 (i.e. we are dealing with a non-0 dimensional problem, see the corresponding chapter) and the name of the result file has not been specified
- $k = -1000$: the value of the flag `Single_Bisection` is not correct
- $k = -3000$: we use the full bissection mode and the problem has more than 10 unknowns

As for the debug option they have been presented in the section 2.3.4.9.

2.5.3 Examples

2.5.3.1 Example 2

The problem we want to solve has been presented in section 2.3.5.2 and section 15.1.1. The name of the test program is `Test_Solve_JH_General1`.

If `epsilonf`=0.01 the program using the maximum equation ordering find the two solutions using 50 boxes.

If we use the single bisection smear mode the program using the maximum equation ordering find the two solutions in 4260ms using 32 boxes.

Note that we may improve the efficiency of the procedure by using simplification procedures such as the 2B (section 2.17) and the 3B method. In that case for `epsilonf`=1e-6, `epsilon`=1e-6 the number of boxes will have been reduced to 2 and both solutions will be guaranteed. even with a search space of $[-\pi, \pi]$ for θ.

2.5.3.2 Example 3

This example is derived from example 2 and has been presented in section 2.3.5.3. The test program is `Test_Solve_JH_General2`. The `IntervalFunction` and `IntervalGradient` have been presented before. The `IntervalHessian` is:

```
INTERVAL_MATRIX IntervalHessian (int l1,int l2,INTERVAL_VECTOR & in)
{
INTERVAL_MATRIX Hess(1,1);
Hess(1,1)=39320.0+(-3128.0+(-78640.0+43560.0*Cos(in(1)))*
      Cos(in(1)))*Cos(in(1))+(-2692.0+(-15152.0+14400.0*Cos(in(1)))*
      Cos(in(1)))*Sin(in(1));
return Hess;
}
```

A test main program may be written as:

```
INT main()
{
int Iterations,Dimension,Dimension_Eq,Apply_Kanto;
int Num,i,j,order,Stop_First_Sol,precision,niter,nn;
double Accuracy,Accuracy_Variable,eps;
INTERVAL_MATRIX SolutionList(200,3);
INTERVAL_VECTOR TestDomain,F(1),P(1),H(3);
VECTOR TR(1),Residu(1);
INTEGER_VECTOR Is_Kanto(6);

Dimension_Eq=Dimension=1;SetTestDomain (TestDomain);

cerr << "Number of iteration = "; cin >> Iterations;
cerr << "Accuracy on Function = "; cin >> Accuracy;
cerr << "Accuracy on Variable = "; cin >> Accuracy_Variable;
cerr << "Debug Level (0,1,2)="; cin >> Debug_Level_Solve_General_Interval;
cerr << "Order (0,1)="; cin >>order;
cerr << "Stop at first solutions (0,1,2)=";cin>>Stop_First_Sol;
cerr << "Apply Kanto (0,1)=";cin>>Apply_Kanto;

Num=Solve_General_JH_Interval(Dimension,Dimension_Eq,
      IntervalTestFunction,IntervalGradient,IntervalHessian,
      TestDomain,order,Iterations,Stop_First_Sol,Accuracy_Variable,
      Accuracy,SolutionList,Is_Kanto,Apply_Kanto,6);

if(Num== -1){cout << "The procedure has failed (too many iterations)"<<endl;return 0;}
cout << Num << " solution(s)" << endl;
for(i=1;i<=Num;i++)
  {
    cout<<"solution "<<i<<endl;cout<<"teta="<<SolutionList(i,1)<<endl;
    cout << "Function value at this point" <<endl;F(1)=SolutionList(i,1);
    cout << Compute_Interval_Function_Gradient(Dimension,Dimension_Eq,
     IntervalTestFunction,IntervalGradient,
```

```
            IntervalHessian,F,1) << endl;
    cout << "Function value at middle interval" <<endl;
    P(1)=Mid(SolutionList(i,1));    F=IntervalTestFunction(1,Dimension_Eq,P);
    cout << Sup(F(1)) << endl;    TR(1)=Mid(SolutionList(i,1));
    if(Is_Kanto(i)==1)cout << "This solution is Kanto" <<endl;
    else cout << "This solution is not Kanto" << endl;
    if(Kantorovitch(Dimension,IntervalTestFunction,IntervalGradient,
                         IntervalHessian,TR,&eps)==1)
      {
P(1)=INTERVAL(TR(1)-eps,TR(1)+eps);
cout << "Unique solution in: " <<P << endl;
      }
    if(Is_Kanto(i)==1)
      {
      nn=Newton(Dimension,IntervalTestFunction,IntervalGradient,TR,Accuracy,1000,Residu);
      if(nn>0)
        {
          cout << "Newton iteration converge toward: " << endl;
          cout << TR << "with residu= " << Residu<< endl;
        }
      else
        {
          if(nn==0)cout << "Newton does not converge" << endl;
          if(nn== -1)cout<<"Newton has encountered a singular matrix"<<endl;
        }
      }
  }
  return 0;
}
```

With `epsilonf`=0.0001 and `epsilon`=0.001 , using Kantorovitch at level 1, we get the solution intervals, using 4 boxes:

$$\theta = [-0.04244333, 0.1295874] \quad \theta = [-0.8376338, -0.7968275]$$

Newton method initialized with the center of these boxes converge toward 4.08282e-15 and -0.8067834.

2.5.3.3 Example 4

In this example (see section 15.1.3) we deal with a complex problem of three equations in three unknowns ψ, θ, ϕ. We are looking for a solution in the domain:

$$[4.537856054, 4.886921908], [1.570796327, 1.745329252], [0.6981317008, 0.8726646262]$$

The system has a solution which is approximatively:

$$4.6616603883, 1.70089818026, 0.86938888189$$

This problem is extremely ill conditioned as for the `TestDomain` the equations intervals are:

$$[-1.45096e+08, 1.32527e+08]; [-38293.3, 29151.5]; [-36389.1, 27705.7]$$

The name of the test program is `Test_Solve_JH_General`.

If we use `epsilonf`=0.1 and `epsilon`=0, we get the first solution with the following number of boxes and computation time:

	full bisection	smear bisection
direct storage	71, 31200ms	38, 20220ms
reverse storage	35, 14710ms	38, 20280ms

We need 71 boxes in the full bisection mode to determine that this solution is unique and 38 boxes in the smear bisection mode.

Note that we may improve the efficiency of the procedure by using simplification procedures such as the 2B (section 2.17) and the 3B method. An interesting point in this example is that the bisection mode 1 (bisecting along the variable whose interval has the largest diameter) is more effective than using the bisection mode 2 (using the smear function) with 33 boxes against 38 for the mode 2 for `epsilonf`=1e-6 and `epsilon`=1e-6. This can easily been explained by the complexity of the Jacobian matrix elements that leads to a large overestimation of their values when using interval: in that case the smear function is not very efficient to determine which variable has the most influence on the equations. But it must be noted that the use of the Hessian allows to reduce this overestimation and consequently the differences in term of used boxes between the two bisection mode is slightly reduced compared to the one we have observed when using only the Jacobian (see section 2.4.3.4).

2.6 Stopping the general solving procedures

It may be interesting to stop the solving procedures although the algorithm has not been completed. We have already seen that a possible mean to do that was to specify a number of roots in such way that the procedure will exit as soon it has found this number of roots.

Another possible way to stop the calculation is to use a time-out mechanism. For that purpose you may define in the double `ALIAS_TimeOut` the maximum number of minutes allowed for the calculation. If this number is reached (approximatively) the procedure will exit and will set the flag `ALIAS_TimeOut_Activated` to 1.

The solution that have been found by the algorithms are stored in the interval matrix `ALIAS_Solution` and their number is `ALIAS_Nb_Solution`. Note that for the procedures involving the Jacobian this matrix will usually describes the boxes that include a unique solution.

2.7 Ridder method for solving one equation

2.7.1 Mathematical background

Ridder method is an iterative scheme used to obtain one root of the equation $F(x) = 0$ within an interval $[x_1, x_2]$. It assumes that $F(x_1)F(x_2) < 0$. Let x_3 be the mid-point of the interval $[x_1, x_2]$. A new estimate of the root is x_4 with:

$$x_4 = x_3 + (x_3 - x_1)\frac{\mathrm{sign(F(x_1) - F(x_2))F(x_3)}}{\sqrt{F(x_3)^2 - F(x_1)F(x_2)}}$$

under the assumption $F(x_1)F(x_2) < 0$ it may be seen that x_4 is guaranteed to lie within the interval $[x_1, x_2]$. As soon as x_4 as been determined we choose as new $[x_1, x_2]$ the interval $[x_1, x_4]$ if $F(x_1)F(x_4) < 0$ or $[x_4, x_2]$ if $F(x_2)F(x_4) < 0$. The convergence of this algorithm is quadratic.

2.7.2 Implementation

Ridder's method enable to find a root of an equation $F(x) = 0$ as soon as the root is bracketed in an interval $[x_1, x_2]$ such that $F(x_1)F(x_2) < 0$. It is implemented as:

```
int Ridder(REAL (* TheFunction)(REAL),INTERVAL &Input,
      double AccuracyV,double Accuracy,int Max_Iter,double *Sol, double *Residu)
```

with:

- `TheFunction`: a procedure which enable to compute the value of the equation at a given point
- `Input`: the interval $[x_1, x_2]$ in which we are looking for a root
- `AccuracyF`: a threshold on the minimal value of the width of the interval $[x_k, x_{k+1}]$ with $F(x_k)F(x_{k+1}) < 0$ considered during the procedure
- `Accuracy`: a threshold on the value of $F(x)$ which determine a root of the equation
- `MaxIter`: maximal number of iteration
- `Sol`: on success the value of the root
- `Residu`: the value of the equation at `Sol`

The procedure returns:

- 1: a solution has been found as `F(Sol)<Accuracy`
- 2: a solution has been found as $|x_k - x_{k+1}|$ <`AccuracyV`
- -1: $F(x_1)F(x_2) > 0$
- -2: a numerical error was encountered during the computation
- -3: the maximal number of iteration has been reached without finding a solution

The test program `Test_Ridder2` present a program to solve the trigonometric equation presented as example 2 (see section 15.1.1).

2.8 Brent method for solving one equation

2.8.1 Mathematical background

Brent method is an iterative scheme used to obtain one root of the equation $F(x) = 0$ within an interval $[x_1, x_2]$. It assumes that $F(x_1)F(x_2) < 0$. Let x_3 be the mid-point of the interval $[x_1, x_2]$. A new estimate of the root is x_4 with:

$$x_4 = x_3 + \frac{P}{Q}$$

with:

$$\begin{aligned} R &= \frac{F(x_3)}{F(x_2)} \quad S = \frac{F(X_3)}{F(x_1)} \quad T = \frac{F(x_1)}{F(x_2)} \\ P &= S[T(R-T)(x_2 - x_3) - (1-R)(x_3 - x_1)] \\ Q &= (T-1)(R-1)(S-1) \end{aligned}$$

In this method x_3 is considered to be the current estimate of the solution. The term P/Q is a correction factor: when this factor leads to a new estimate of the solution outside the interval we use a bisection method to compute a new interval $[x_1, x_2]$. In other words if $F(x_1)F(x_3) < 0$ the new interval is $[x_1, x_3]$ and if $F(x_2)F(x_3) < 0$ the new interval is $[x_3, x_2]$. Therefore Brent method is a cross between a bisection method and a super-linear method which insure that the estimate of the solution always lie within the interval $[x_1, x_2]$.

2.8.2 Implementation

Brent's method enable to find a root of an equation $F(x) = 0$ as soon as the root is bracketed in an interval $[x_1, x_2]$ such that $F(x_1)F(x_2) < 0$. It is implemented as:

```
int Brent(REAL (* TheFunction)(REAL),INTERVAL &Input,
      double AccuracyV,double Accuracy,int Max_Iter,double *Sol, double *Residu)
```

with:

- `TheFunction`: a procedure which enable to compute the value of the equation at a given point
- `Input`: the interval $[x_1, x_2]$ in which we are looking for a root
- `AccuracyF`: a threshold on the minimal value of the width of the interval $[x_k, x_{k+1}]$ with $F(x_k)F(x_{k+1}) < 0$ considered during the procedure
- `Accuracy`: a threshold on the value of $F(x)$ which determine a root of the equation
- `MaxIter`: maximal number of iteration
- `Sol`: on success the value of the root
- `Residu`: the value of the equation at `Sol`

The procedure returns:

- 1: a solution has been found as `F(Sol)<Accuracy`
- 2: a solution has been found as $|x_k - x_{k+1}|$ <`AccuracyV`
- -1: $F(x_1)F(x_2) > 0$
- -3: the maximal number of iteration has been reached without finding a solution

The test program `Test_Ridder2` present a program to solve the trigonometric equation presented as example 2 (see section 15.1.1).

2.9 Newton method for solving systems of equations

2.9.1 Mathematical background

Let $\mathcal{F}$ be a system of n equations in the n unknowns $\mathbf{x}$ and $\mathbf{x_0}$ be an estimate of the solution of the system. Let J be the Jacobian matrix of the system of equation. Then the iterative scheme defined by:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + J^{-1}(\mathbf{x_k})\mathcal{F}(\mathbf{x_k}) \tag{2.6}$$

starting with $\mathbf{x_0}$ may converge toward a solution of the system.

A *simplified Newton method* consist in using a constant matrix in the classical Newton method, for example the inverse Jacobian matrix at some point like $\mathbf{x_0}$. The iterative scheme become:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + J^{-1}(\mathbf{x_0})\mathcal{F}(\mathbf{x_k}) \tag{2.7}$$

Although the simplified method may need a larger number of iteration before converging than the classical scheme each iteration has a lower computation time as there is no computation of the inverse of the Jacobian matrix. This method may also encounter convergence problem as it has a convergence ball smaller than the classical Newton method.

Newton method has advantages and drawbacks that need to be known in order to use it in the best way:

- it may really be fast: this may be important, for example in real-time control
- it is very simple to use
- **but** it does not necessarily converge toward the solution "closest" to the estimate (see the example in section 15.1.2)
- **but** it may not converge. Kantorovitch theorem (see section 3.1.2) enable to determine the size of the convergence ball but this size is usually small (but quite often in practice the size is greater than the size given by the theorem which however is exact in some cases)
- **but** a numerical implementation of Newton may overflow

2.9.2 Implementation

The procedure for using Newton method is:

```
int Newton(int n,VECTOR (* TheFunction)(VECTOR &),
        MATRIX (* Gradient)(VECTOR &),
        VECTOR &Input,double Accuracy,int MaxIter,VECTOR &Residu)
```

with

- `n`: number of equations
- `TheFunction`: a procedure which return the value of the equation for given values of the unknowns (see note 2.3.4.3)

- `Gradient`: a procedure which return the Jacobian matrix of the system for given values of the unknowns (see note 2.4.2.2)
- `Input`: at the start of the procedure the estimate of the solution, at the end of the procedure the solution
- `Accuracy`: the procedure return a solution if there is an `Input` such that $|\mathcal{F}_k($`Input`$)| \leq$ `Accuracy` for all k in $[1,n]$.
- `MaxIter`: the procedure will return a failure code if a solution is not found after `MaxIter` iteration
- `Residu`: the value of the equations for the solution

Note that it also possible to use in the Newton method the interval evaluation of the equation and of the Jacobian matrix which are necessary for the general purpose solving algorithm with Jacobian (see section 2.4). The syntax of this implementation is:

```
int Newton(int Dimension,
        INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
        INTERVAL_MATRIX (* IntervalGradient)(int, int, INTERVAL_VECTOR &),
        VECTOR &Input,double Accuracy,int MaxIter,VECTOR &Residu)
```

- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see the note 2.3.4.3
- `IntervalGradient`: a function which the interval matrix of the jacobian of the equations, see the note 2.4.2.2

To avoid overflow problem it is possible to use the vector `ALIAS_Newton_Max_Dim` that must be resized to the number of unknowns and in which will be indicated the maximal possible value of each variable after each Newton operation. If one of these values is exceeded `Newton` will return 0.

The version of Newton method with constant J^{-1} matrix is implemented as:

```
int Newton(int n,VECTOR (* TheFunction)(VECTOR &),
        MATRIX &InvGrad,VECTOR &Input,double Accuracy,int MaxIter,VECTOR &Residu)
```

There is a special implementation of Newton method for univariate polynomial P:

```
int Newton(int Degree,REAL *Input,VECTOR &Coeff,double Accuracy,int Max_Iter,REAL *Residu)
```

with:

- `Degree`: degree of the polynomial
- `Input`: on entry an estimate of the solution and on exit the solution
- `Coeff`: coefficient of the polynomial ordered in increasing degree
- `Accuracy`: `Input` is a solution if —$P($`Input`—$\leq$ `Accuracy`
- `MaxIter`: the procedure will return a failure code if a solution is not found after `MaxIter` iteration
- `Residu`: the value of the polynomial for the solution

In that case we may have a problem if the `Accuracy` cannot be reached due to numerical errors. If you have determined that Newton should converge (using for example Kantorovitch theorem, see section 3.1.2) then you may use the procedure `Newton_Safe` with the same argument: this procedure will return the solution which has led to the lowest `Residu` during the Newton scheme.

An example of use of the Newton method is presented in section 15.1.1, where it is compared to alternative methods.

2.9.2.1 Return value

- -1: A singular matrix has been found during the scheme (not applicable if we use Newton with a constant J^{-1})
- 0: Newton has not converged after `MaxIter` iteration
- 1: Newton has converged toward solution `Input`
- 2: only valid for the implementation in which the function evaluation return an interval vector. It has not been possible to find a solution such that all the mid point of the interval evaluation of the function was at a distance less than `Accuracy` from 0. However the procedure will return as solution the point obtained during Newton iteration for which the interval evaluation of all the function include 0 and has the minimal average value for the width of the interval evaluation of the functions.

2.9.2.2 Functions

The procedures `TheFunction` and `IntervalFunction` should be user written. They return the value of the equations (either as a vector of `REAL` or as a vector of `INTERVAL`, see 2.3.4.3) for given values or intervals for the unknowns. They take one argument which is the vector of `REAL` which describe the unknowns.

In the same way the procedures `Gradient` and `IntervalGradient` should be user written. They return the Jacobian matrix of the system of equations of the equations (either as a matrix of `REAL` or as a matrix of `INTERVAL`) for given values of the unknowns. The `Gradient` procedure take one argument which is the vector of `REAL` which describe the unknowns. The `IntervalGradient` procedure has three arguments and is described in section 2.4.2.2.

2.9.3 Systematic use of Newton

It may be interesting to systematically use the Newton scheme in a solving procedure in order to quickly determine the solutions of a system of equations.

For that purpose we may use the `TryNewton` procedure whose purpose is to run a few iterations of the Newton scheme for a given box. The syntax of this procedure is:

```
int TryNewton(int DimensionEq,int DimVar,
              INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
              INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
              INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),
              double Accuracy,
              int MaxIter,
              INTERVAL_VECTOR &Input,
              INTERVAL_VECTOR &Domain,
              INTERVAL_VECTOR &UnicityBox)
```

where

- `DimensionEq`: number of equations
- `DimVar`: number of variables
- `TheIntervalFunction`: a procedure in `MakeF` format for computing an interval evaluation of the equations
- `Gradient`: a procedure that compute the jacobian in `MakeJ` format
- `Hessian`: a procedure in `MakeH` format that computes the Hessian of the system
- `Domain`: the domain in which we are looking for solutions of the system
- `Input`: a sub-box of `Domain`

The mid-point of `Input` is used as initial guess of the Newton scheme. The parameters `Accuracy` is used in the Newton scheme to determine if Newton has converged i.e. if the residues are lower than `Accuracy`. A maximum of `MaxIter` iterations are performed.

If the Newton scheme converges, the presence of a single solution in the neighborhood of the approximated solution is checked by using the Kantorovitch theorem (see section 3.1.2). If this check is positive, then a ball that includes this single solution is determined and returned in `UnicityBox`. If the flag `ALIAS_Epsilon_Inflation` is set to 1, then the inflation scheme is used to try to enlarge this unicity box.

This procedure returns 11 if an unicity box has been determined, 0 otherwise. Note that this procedure is already embedded in `HessianSolve`.

2.10 Krawczyk method for solving systems of equations

2.10.1 Mathematical background

Let $\mathcal{F}$ be a system of n equations in the n unknowns $\mathtt{x}$. Let $\mathbf{X}$ be a range vector for $\mathbf{x}$ and $y_0 = Mid(\mathbf{X})$. Let r_0 be the norm of the matrix $I - YF'(\mathbf{X})$. Let the following iterative scheme for $k \geq 1$:

$$\begin{aligned}
&y_k = Mid(\mathbf{X}_k)\\
&Y_k = \begin{cases} (Mid(F'(\mathbf{X}_k))^{-1} & \text{if} \;\; ||I - YF'(\mathbf{X}_k)|| \leq r_{k-1} \\ Y_{k-1} & \text{otherwise} \end{cases}\\
&r_k = ||I - Y_k F'(\mathbf{X}_k)||
\end{aligned}$$

Let define K as:

$$K\mathbf{X} = y - YF(y) + \{I - YF'(\mathbf{X})\}(\mathbf{X} - y)$$

If

$$K(\mathbf{X}_0) \subseteq \mathbf{X}_0 \quad \text{and} \quad r_0 < 1$$

then the previous iterative scheme will converge to the unique solution of F in $\mathbf{X}$ [8]. The procedure described in section 3.1.1 enable to verify if the scheme will be convergent.

2.10.2 Implementation

The procedure is implemented as:

```
int Krawczyk_Solver(int m,int n,
        INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
        INTERVAL_MATRIX (* IntervalGradient)(int, int, INTERVAL_VECTOR &),
        INTERVAL_VECTOR &Input,double Acc , VECTOR &Result)
```

with

- `m`: the number of unknown
- `n`: the number of equations
- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see the note 2.3.4.3
- `IntervalGradient`: a function which the interval matrix of the jacobian of the equations, see the note 2.4.2.2
- `Input`: the ranges for the variables
- `Acc`: the algorithm will return the result if $|F_i| \leq$ `Acc`
- `Result`: the solution of the system

The procedure will return 1 if it has converged to a solution, 0 or -1 otherwise.

2.11 Solving univariate polynomial with interval analysis

2.11.1 Mathematical background

Clearly interval analysis may be used for solving univariate polynomial, especially if we are looking for some roots within a specific interval (note that for generic polynomial we may always obtain intervals in which lie the positive and negative roots using the algorithm described in section 5.2).

2.11.2 Implementation

The algorithm we have implemented is a direct derivation of the general purpose solving algorithm with Jacobian and Hessian in which these values are automatically derived. To isolate the roots we use the Kantorovitch theorem (which may also optionally be used during the resolution, see section 3.1.2). To eliminate boxes during the bisection process we use the safe Budan-Fourier method (see section 5.5.2).

```
int Solve_UP_JH_Interval(int Degree,VECTOR Coeff,
            INTERVAL & TheDomain,
            int Order,int M,int Stop,
            double epsilon,double epsilonf,
            INTERVAL_VECTOR & Solution,
            INTEGER_VECTOR & IsKanto,int NbSolution);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `TheDomain`: the interval in which we are looking for roots
- `Order`: the type of order which is used to store the intervals created during the bisection process. This order may be either `MAX_FUNCTION_ORDER` or `MAX_MIDDLE_FUNCTION_ORDER`. See the note on the order 2.3.4.4.
- `M`: the maximum number of boxes which may be stored. See the note 2.5.2.2
- `Stop`: the possible values are 0,1,2
  - 0: the algorithm will look for every solution in `TheDomain`
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found
- `epsilon`: the maximal width of the box, see the note 2.3.4.6
- `epsilonf`: the maximal width of the equation intervals, see the note 2.3.4.6
- `Solution`: an interval matrix of size (`Nb`,`m`) which will contained the solution intervals.
- `IsKanto`: an integer vector of dimension `Nb`. A value of 1 for `IsKanto(i)` indicate that Newton method (see section 2.9) with as estimate the center of some solution interval `Solution(i)` has been used and has converged toward the unique solution `Solution(i)` which lie within this solution intervals. Note that the interval which contain the solution may be retrieved in the interval vector `Interval_Solution_UP`.
- `NbSolution`: the maximum number of solution we are looking for.

The procedure will return the number of solution(s) or -3 if the order is not 0 or 1, -2 if the number of equations or unknowns is equal or lower to 0 and -1 if the number of iteration is too low. There is an alternate form of this procedure in the case where we are looking for all the roots of the polynomial.

```
int Solve_UP_JH_Interval(int Degree,VECTOR Coeff,
            int Order,int M,int Stop,
            double epsilon,double epsilonf,
            INTERVAL_VECTOR & Solution,
            INTEGER_VECTOR & Is_Kanto,int NbSolution);
```

There are two alternate forms of this procedure in the case where we are looking for the positive or negative roots of the polynomial.

```
int Solve_UP_JH_Positive_Interval
int Solve_UP_JH_Negative_Interval
```

In the three previous procedures there is no `TestDomain` as it is automatically determined by the procedure. If there was a failure in the determination of the domain (for the reasons explained in section 5.2) the procedures will return -1.

The previous procedures are numerically safe in the sense that we take into account rounding errors in the evaluation of the polynomial and its gradient. For well conditioned polynomials you may use faster procedures whose name has the prefix `Fast`. For example `Fast_Solve_UP_JH_Interval` is the general procedure for finding the roots of a polynomial.

Clearly this procedure is not intended to be used as substitute to more classical algorithms.

It makes use of a specific Krawczyk procedure for polynomials:

```
int Krawczyk_UP(int Degree,INTERVAL_VECTOR &Coeff,
             INTERVAL_VECTOR &CoeffG,INTERVAL &Input)
```

2.11.2.1 Example

The program `Test_Solve_UP` is a general test program which enable to solve univariate polynomial whose coefficients are given in a file by increasing power of the unknown.

We use as example the Wilkinson polynomial of degree n where P:

$$P = \prod_{i=1}^{i=n}(x-i)$$

It is well known that this polynomial is extremely ill-conditioned. For $n = 12$ the coefficient of x^{11} is 78. But if we modify this coefficient by 10^{-5} there is a big change in the roots, 4 of them becoming complex [4]. The general procedure leads to reasonable accurate result up to $n = 18$. At $n = 19$ although Kantorovitch theorem has determined interval solutions that indeed contain all the solutions, Newton method is unable to provide an accurate estimate of this root due to numerical errors.

For $n = 10$ and if we are looking for the roots in the interval [0,2] the computation time is 90ms, for $n = 15$ 190ms and 330ms for $n = 20$. For the fast algorithm these times are: 10ms, 20ms, 30 ms Note that the best classical solving algorithm start to give inaccurate results for $n = 22$ (between 12.5 and 18.5 the interval analysis algorithm finds the roots 13.424830, 13.538691, 15.477653, 15.498664, 17.554518, 17.553513) and give imaginary roots for $n = 23$.

2.12 Solving univariate polynomial numerically

As an alternative to interval solving `ALIAS` proposes a numerical algorithm

```
ALIAS_Solve_Poly((double *C, int *Degree,double *Sol),
```

The arguments are the coefficients `C` of the polynomial, a pointer to the integer `Degree` that is initially the degree of the polynomial and `Sol` which will be used to store the real roots. This procedure returns the number of real roots or `-1` if the computation has failed. The procedure `ALIAS_Solve_Poly_PR` takes the same arguments but returns the real part of the roots.

There is also a version of the Newton scheme for univariate polynomial:

```
int Fast_Newton(int Degree,REAL *Input,VECTOR &Coeff,VECTOR &CoeffG,
          double Accuracy,int Max_Iter,REAL *Residu)
```

where

- `Degree` is the degree of the polynomial
- `Input`: an approximation of the solution
- `Coeff`: the polynomial is written as `Coeff[1]+Coeff[2]x+.....Coeff[Degree+1]pow(x,Degree)`
- `CoeffG`: the coefficients of the derivative of the polynomial
- `Accuracy`: the algorithm will stop if the absolute value of the evaluation of the polynomial at the current point is lower than this number
- `Max_Iter`: maximum number of iteration
- `Residu`: the value of the polynomial at the solution

This procedure returns -1 if the derivative polynomial is 0. Otherwise it returns 1 if a solution has been found or 0 if `Max_Iter` iteration has been completed.

2.13 Solving trigonometric equation

2.13.1 Mathematical background

The purpose of this section is to present an algorithm which enable to determine the roots of an equation F in the unknown x of the form:

$$F = \sum a_k \sin^m(x) \cos^n(x) \tag{2.8}$$

with m in $[0, M]$ and n in $[0, N]$, m, n being integers. We use the half angle tangent substitution. If θ is the unknown we define T as:

$$T = \tan(\frac{\theta}{2})$$

Then we have:

$$\sin(\theta) = \frac{2T}{1+T^2} \qquad \cos(\theta) = \frac{1-T^2}{1+T^2}$$

Note that the change of variable is not valid if $\theta = \pm\pi$. In that case it will be preferable to define $\alpha = \theta + \pi$ and to transform the initial into an equation in α. Then the change of variable may be applied.

Using the above relation any trigonometric equation can be transformed into a polynomial equation which is solved using the tools of section 2.11.

It remains to define an interval for angles that we will denote an *angle interval*. The element of an angle interval is usually defined between 0 and 2π (although in most of the following procedures any value can be used when not specified: internally the element of the angle interval are converted into value within this range). A difference between numbers interval (`INTERVAL`) and angle interval is that the lower bound of an angle interval may be larger than the upper bound. Indeed the order in an angle interval is important: for example the angle intervals [0,π/4] and [π/4,0] are not the same.

2.13.2 Implementation

The purpose of this procedure is to determine the roots of a trigonometric equation within a given angle interval.

```
int Solve_Trigo_Interval(int n,VECTOR &A,INTEGER_VECTOR &SSin,
           INTEGER_VECTOR &CCos,double epsilon,double epsilonf,
      int M,int Stop,INTERVAL_VECTOR &Solution,int Nb,REAL Inf,REAL Sup);
```

with:

- `M`: the maximum number of boxes which may be stored. See the note 2.3.4.5
- `Stop`: the possible values are 0,1,2
  - 0: the algorithm will look for every solution in the angle interval
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found
- `epsilon`: the maximal width of the box, see the note 2.3.4.6
- `epsilonf`: the maximal width of the equation intervals, see the note 2.3.4.6
- `Solution`: an interval vector of size at least `Nb` which will contained the solution intervals.
- `Nb`: the maximal number of solution which will be returned by the algorithm
- `Inf, Sup`: the bound of the angle interval in which we are looking for solutions.

Note that the returned `Solution` will always be a range `[a,b]` included in $[0,2\pi]$ and in `[Inf,Sup]`, this interval angle being reduced to an angle interval in the range $[0,2\pi]$. The procedure will return:

- ≥ 0: the number of roots
- `-1`: the bound `Inf` or `Sup` is incorrect (positive or negative infinity)

If you are looking for all the roots of the trigonometric equation you may either specify `[Inf,Sup]` as $[0,2\pi]$ or use the procedure:

```
int Solve_Trigo_Interval(int n,VECTOR &A,INTEGER_VECTOR &SSin,
            INTEGER_VECTOR &CCos,double epsilon,double epsilonf,
            int M,int Stop,INTERVAL_VECTOR &Solution,int Nb);
```

This procedure first analyze the trigonometric equation to find bounds on the roots using the algorithm described in section 4.3, then use the previous procedure to determine the roots within the bound. In some case this procedure may be faster than the general purpose algorithm.

2.13.3 Examples

The test program `Test_Solve_Trigo` enable to determine the roots of any trigonometric equation which is described in a file. In this you indicate first the coefficient of the term, its sine power and then its cosine power, this for each term of the equation.

We consider the trigonometric equation derived in section 15.1.2:

$$-508\sin(\theta)-25912\cos(\theta)+3788\cos(\theta)\sin(\theta)+11092+19660\,(\cos(\theta))^2-1600\,(\cos(\theta))^2\sin(\theta)-4840\,(\cos(\theta))^3=0$$

which has 0,5.47640186917958647 as roots. The general procedure find the roots 1.743205553711625e-11, 5.476401869153828 while the procedure using the determination of the bounds (which are [0,1.570796326794896], [5.235987755982987,6.283185307179586]) find the roots 5.974250609803587e-12, 5.476401869153828.

2.14 Solving systems with linear and non-linear terms: the simplex method

2.14.1 Mathematical background

Consider a system of n equations $F_1,\dots,F_n$ in the m unknowns $X=\{x_1,\dots,x_m\}$ such that the F_i may be written as:

$$F_i=G_i(X)+a_0+\sum a_jx_j$$

F_i is the sum of the non-linear function G_i and of the linear terms with coefficients a_j. Let the interval evaluation of G_i be $[\underline{G_i}, \overline{G_i}]$: we define a new variable Y_i as $Y_i = G_i - \underline{G_i}$, which imply $Y_i \geq 0$. F_i may now be written as the sum of linear term:

$$F_i = Y_i + a_0 + \underline{G_i} + \sum a_j x_j$$

Hence the system is now a linear system with the additional constraint that $Y_1 \geq 0, \ldots, Y_n \geq 0$. We may now apply a well-known method in linear programming: the *simplex* method which can be used to find the minimum or maximum of a linear function under the m_1 equality constraints $G_1(X) = 0, \ldots, G_{m_1}(X) = 0$ and the m_2 inequality constraints $K_1(X) \geq 0, \ldots, K_{m_2}(X) \geq 0$. There are two phases in the simplex method: phase I verifies if a feasible solution exists and phase II is used to determine the extremum of the linear function.

In our case we may use only phase I or phase I and II by considering the $2m$ optimum problems which are to determine the minimum and maximum of the m unknowns under the $2n$ constraints $F_i(X) = 0, Y_i(X) = 0$ and update the interval for an unknown if the simplex applied to minimize or maximize enable to improve the range. It may be seen that this is a recursive procedure: an improvement on one variable change the constraint equations and may thus change the result of the simplex method applied for determining the extremum of a variable which has already been considered.

This procedure, proposed in [25], enable to correct one of the drawback of the general solving procedures: each equation is considered independently and for given intervals for the unknowns two equations may have an interval evaluation that contain 0 although these equations cannot be canceled *at the same time*. The previous method enable to take into account at least partly the dependence of the equations. Clearly it will more efficient if the functions has a large number of linear terms and a "small" non-linear part.

In all of the following procedures the various storage mode and bisection mode of the general solving procedures may be used and inequalities are handled.

2.14.2 Implementation without gradient

The procedure is implemented as:

```
int Solve_Simplex(int m,int n,int NbNl,
        INTEGER_VECTOR TypeEq,
        INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
        void (* NonLinear)(INTERVAL_VECTOR &x,INTERVAL_VECTOR &X),
        void (* CoeffLinear)(MATRIX &U),
        double MaxDiam,
        int FullSimplex,
        INTERVAL_VECTOR & TheDomain,
        int Order,int Iteration,int Stop,
        double epsilon,double epsilonf,double Dist,
        INTERVAL_MATRIX & Solution,int Nb,
        int (* Simp_Proc)(INTERVAL_VECTOR &))
```

the arguments being:

- `m`: number of unknowns
- `n`: number of equations, see the note 2.3.4.1
- `NbNl`: number of equations that have no linear term at all or are inequalities. If you omit this parameter its value will be assumed to be 0 and you have to omit the `TypeEq` parameter.
- `TypeEq`: an array of integers that indicate the type for the equations. `TypeEq(i)` is -1,0,1 if equation `i` is an inequality ≤ 0, an equation or an inequality ≥ 0.
- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see the note 2.3.4.3 on how to write this procedure. The equations must be ordered: first the equations with linear terms then the equations without any linear terms and finally the inequalities
- `NonLinear`: a procedure to compute the non linear part of the equations, see note 2.14.2.1

- `CoeffLinear`: a procedure that return a matrix which contain the constant coefficients of the linear term in the equation, see note 2.14.2.2
- `MaxDiam`: the simplex method will not be used on boxes whose maximal width is lower than this value. Should be set to 0 or a small value
- `FullSimplex`: this flag is used to indicate how much we will use the simplex method (which may be costly). If set to -1 only the phase I of the simplex will be used. If set to k with $k \geq 0$, then the full simplex method will be used recursively on the $k+1$ variables having the largest interval width.
- `TheDomain`: box in which we are looking for solution of the equations
- `Order`: the type of order which is used to store the intervals created during the bisection process. This order may be either `MAX_FUNCTION_ORDER` or `MAX_MIDDLE_FUNCTION_ORDER`. See the note on the order 2.3.4.4.
- `M`: the maximum number of boxes which may be stored. See the note 2.3.4.5
- `Stop`: the possible values are 0,1,2
  - 0: the algorithm will look for every solution in `TheDomain`
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found
- `epsilon`: the maximal width of the solution intervals, see the note 2.3.4.6
- `epsilonf`: the maximal width of the equation intervals for a solution, see the note 2.3.4.6
- `Dist`: minimal distance between the middle point of two interval solutions, see the note 2.3.4.7
- `Solution`: an interval matrix of size (`Nb`,`m`) which will contained the solution intervals. This list is sorted using the order specified by `Order`
- `Nb`: the maximal number of solution which will be returned by the algorithm
- `Simp_Proc`: a user-supplied procedure that take as input the current box and proceed to some further reduction of the width of the box or even determine that there is no solution for this box, in which case it should return -1 (see note 2.3.3). Remember that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2).

Note that the `Simp_Proc` argument may be omitted.

2.14.2.1 The `NonLinear` procedure

The purpose of the procedure is to compute an interval evaluation of the non linear term of the equations (the G_i terms in the mathematical background). This procedure take as argument a box and the returned evaluation of the non-linear terms. Consider for example the equations:

$$-0.1x_1^2 + 0.25x_1^3 + x_1 + 0.5x_2 + 0.3 = 0$$
$$-0.1x_2^2 + 0.55x_2^3 + 2x_1 - 0.5x_2 + 0.2 = 0$$

The non-linear terms of these equations are $-0.1x_1^2+0.25x_1^3$ and $-0.1x_2^2+0.55x_2^3$ and the `NonLinear` procedure is written as:

```
void NonLinear(INTERVAL_VECTOR &x,INTERVAL_VECTOR &X)
{
    X(1) = (-0.1+0.25*x(1))*Sqr(x(1));
    X(2) = (-0.1+0.55*x(2))*Sqr(x(2));
}
```

2.14.2.2 The `CoeffLinear` procedure

This procedure returns a matrix U of dimension $\mathtt{n} \times \mathtt{m}+1$ with $U(i,j)$ equal to the coefficient of x_j in the equation i while $U(i,\mathtt{m}+1)$ is the constant term of the equation. In the previous example this procedure will be:

```
void CoeffLinear(MATRIX &U)
{
U(1,1)=1; U(1,2)=0.5; U(1,3)=0.3;
U(2,1)=2; U(1,2)=-0.5; U(1,3)=0.2;
}
```

2.14.2.3 Using an expansion

In some case it may be interesting to consider an expansion of the function around a given point. For example consider the term x^2 with x in the range [1,2]: in the simplex method the range [1,4], for this term will be added to the non linear part of the equation. But if we substitute the unknown x by a new unknown x_1 such that $x = 1 + x_1$ (hence the range for x_1 will be [0,1]) we will get $x^2 = 1 + 2x_1 + x_1$ we will get an additional linear term $(2x_1)$ while the non linear part will be x_1^2 with the range [0,1]. For each variable y_i in the range $[\underline{y_i}, \overline{y_i}]$ we may define a new variable y_i^1 such that $y_i = \underline{y_i} + y_i^1$ where y_i^1 has the range $[0, \overline{y_i} - \underline{y_i}]$. We may then write the non linear and linear procedures for the unknowns y_i^1 but it necessary to notify the simplex procedure that such an expansion is used. This is done by setting the flag `ALIAS_Simplex_Expanded` to 1 (this possibility is available only for the simplex method using the gradient).

You may also prohibit the use of the simplex method in the procedure (for example to use it only in your own simplification procedure) by setting the flag `ALIAS_DONT_USE_SIMPLEX` to 1.

2.14.2.4 Example

The main program for the previous system of 2 equations (omitting the procedure `F` that computes the interval evaluation of the equations) will be written as

```
int main()
{
int num,i;
INTERVAL_MATRIX Solution(100,2);
INTERVAL_VECTOR x(2);
INTEGER_VECTOR Type(2);

Clear(Type);
for(i=1;i<=2;i++)x(i)=INTERVAL(-10,10);
Single_Bisection=1;
num=Solve_Simplex(2,2,0,Type,F,NonLinear,CoeffLinear,0.001,0,x,
  0,40000,0,0.,0.0001,0.0001,Solution,100);

for(i=1;i<=num;i++)
  {
    x(1)=Solution(i,1);x(2)=Solution(i,2);
    cout<<"Solution "<<i<<":"<<x<<endl;
  }
}
```

2.14.3 Implementation with gradient

This procedure may be used if the gradient of the equations are available. A full version is implemented as:

```
int Solve_Simplex_Gradient(int m,int n,int NbNl,
        INTEGER_VECTOR TypeEq,
        INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
        INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
        void (* NonLinear)(INTERVAL_VECTOR &x,INTERVAL_VECTOR &X),
        void (* GradientNonLinear)(INTERVAL_VECTOR &x,INTERVAL_MATRIX &X),
        void (* CoeffLinear)(MATRIX &U),
        double MaxDiam,
        int FullSimplex,
        INTERVAL_VECTOR & TheDomain,
```

```
int Order,int Iteration,int Stop,
double epsilon,double epsilonf,double Dist,
INTERVAL_MATRIX & Solution,
int Nb,int UseGradNL,
INTEGER_MATRIX &GI,
int (* Simp_Proc)(INTERVAL_VECTOR &))
```

the arguments being:

- `m`: number of unknowns
- `n`: number of equations, see the note 2.3.4.1
- `NbNl`: number of equations that have no linear term at all or are inequalities.
- `TypeEq`: an array of integers that indicate the type for the equations. `TypeEq(i)` is -1,0,1 if equation `i` is an inequality ≤ 0, an equation or an inequality ≥ 0.
- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see the note 2.3.4.3 on how to write this procedure. The equations must be ordered: first the equations with linear terms then the equations without any linear terms and finally the inequalities
- `Gradient`: a procedure which return the Jacobian matrix of the system for given values of the unknowns (see note 2.4.2.2)
- `NonLinear`: a procedure to compute the non linear part of the equations, see note 2.14.2.1
- `GradientNonLinear`: a procedure that returns the gradient of the non linear part of the equations, see note 2.14.3.1
- `CoeffLinear`: a procedure that return a matrix which contain the constant coefficients of the linear term in the equation, see note 2.14.2.2
- `MaxDiam`: the simplex method will not be used on boxes whose maximal width is lower than this value. Should be set to 0 or a small value
- `FullSimplex`: this flag is used to indicate how much we will use the simplex method (which may be costly). If set to -1 only the phase I of the simplex will be used. If set to k with $k \geq 0$, then the full simplex method will be used recursively on the $k+1$ variables having the largest interval width.
- `TheDomain`: box in which we are looking for solution of the equations
- `Order`: the type of order which is used to store the intervals created during the bisection process. This order may be either `MAX_FUNCTION_ORDER` or `MAX_MIDDLE_FUNCTION_ORDER`. See the note on the order 2.3.4.4.
- `M`: the maximum number of boxes which may be stored. See the note 2.3.4.5
- `Stop`: the possible values are 0,1,2
  - 0: the algorithm will look for every solution in `TheDomain`
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found
- `epsilon`: the maximal width of the solution intervals, see the note 2.3.4.6
- `epsilonf`: the maximal width of the equation intervals for a solution, see the note 2.3.4.6
- `Dist`: minimal distance between the middle point of two interval solutions, see the note 2.3.4.7
- `Solution`: an interval matrix of size (`Nb`,`m`) which will contained the solution intervals. This list is sorted using the order specified by `Order`

- `Nb`: the maximal number of solution which will be returned by the algorithm
- `UseGradNL`: if set to 1 the algorithm will use the gradient of the non linear part of the equations to improve their interval evaluation. Otherwise must be set to 0.
- `GI`: an integer matrix which give a-priori information on the sign of the derivative of the function. `GI(i,j)` indicates the sign of the derivative of function `i` with respect to variable `j` using the following code:
  - -1: the derivative is always negative
  - 0: the function is not dependent of variable `j`
  - 1: the derivative is always positive
  - 2: the sign of the derivative is not known
- `Simp_Proc`: a user-supplied procedure that take as input the current box and proceed to some further reduction of the width of the box or even determine that there is no solution for this box, in which case it should return -1 (see note 2.3.3). Remember that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2).

The following variables play also a role in the computation:

- `ALIAS_Store_Gradient`: if not 0 the gradient matrix of each box will be stored together with the boxes. Must be set to 0 for large problem (default value: 1)
- `ALIAS_Diam_Max_Gradient`: if the maximal width of the ranges in a box is lower than this value, then the gradient will be used to perform the interval evaluation of the functions (default value: 1.e10)
- `ALIAS_Diam_Max_Kraw`: if the maximal width of the ranges in a box is lower than this value, then the Krawczyk operator will be used to determine if there is a unique solution in the box (default value: 1.e10)
- `ALIAS_Diam_Max_Newton`: if the maximal width of the ranges in a box is lower than this value, then the interval Newton method will be used either to try to reduce the width of the box or to to ensure that there is no solution of the system in the box (default value: 1.e10)
- `Min_Diam_Simplex`: if the maximal width of the input box is lower than this value, then the simplex method will be used
- `Max_Diam_Simplex`: if the maximal width of the input box is lower than this value, then the simplex method will not be used
- `Min_Improve_Simplex`: when applying the simplex method if the change on one variable is larger than this value, then the simplex method will be repeated
- `ALIAS_Simplex_Expanded`: if set to 1 the expression has been expanded with respect to the lower bound of each variable

There are several versions of this procedure in which several arguments of the general procedure may be omitted. The following table indicates which arguments may be omitted and the corresponding assumptions (EO=equations only).

omitted						
`NbNl`		0		0		0
`TypeEq`	EO	EO	EO	EO	EO	EO
`GradientNonLinear`			not known			not known
`UseGradNL`			0	0	0	0

In all cases you may omit the `GI` argument (the derivatives are assumed to be unknown) and `Simp_Proc`.

2.14.3.1 The GradientNonLinear procedure

The purpose of this procedure is to compute the jacobian of the non linear part of the equations in order to improve their interval evaluation. The syntax of this procedure is:

```
void GradientNonLinear(INTERVAL_VECTOR &x,INTERVAL_MATRIX &J)
```

where x is an interval vector which contains the range for the unknowns and J is the corresponding jacobian interval matrix.

2.15 Solving systems with determinants

In some cases systems may involve determinants: for example in algebraic geometry the resultant of two algebraic equations is defined as the determinant of the Sylvester matrix. To get an analytical form of the equations it is therefore necessary to compute the determinant: but in some cases this may be quite difficult as the expression may be quite large, even for sparse matrix. A mechanism in ALIAS enable to get the interval evaluation of an equation which include determinant without having to provide the analytical form of the determinant but only the interval evaluation of the *components* of the matrix. This mechanism is based on procedures that compute the interval evaluation of the determinant of an interval-valued matrix (see section 7.1).

Using these procedures it is possible to design the equation evaluation procedure that are used in the general solving procedure of ALIAS as described in 2.3.4.3. Assume for example that you have to evaluate the expression

$$(x + |A|) * y + 2 * (y + |B|)$$

where A, B are two matrices of dimension 6. Assume that V is an interval vector which contain the interval values for x, y and that you have to return the interval evaluation of this equation in a interval vector W. Then you may write the following procedure:

```
INTERVAL_MATRIX A(6,6),B(6,6);
A=Compute_A(V) //compute A for the interval value of x,y
B=Compute_B(V) //compute B for the interval value of x,y
W(1)=(V(1)+Medium_Determinant(A))*V(2)+2*(V(2)+Medium_Determinant(B));
```

You must be however careful when using this procedure in a denominator as the presence of 0 in the interval evaluation of the determinant is not checked, which will lead to an error when computing the interval evaluation of an equation (see section 2.1.1.3).

Note that the MakeF procedure of the ALIAS-Maple package is able to produce efficient code for an equation file even if unexpanded determinants are present in the equation.

There are also procedures to compute the derivatives of a determinant Note that the MakeJ procedure of the ALIAS-Maple package is able to produce a procedure compatible with the requirements of the gradient procedure required by the library (see section 2.4.2.2) even if determinants are present in the equation.

There are also procedures to compute the second order derivatives of a determinant Note that the MakeH procedure of the ALIAS-Maple package is able to produce a procedure compatible with the requirements of the hessian procedure required by the library (see section 2.5.2.1) even if determinants are present in the equation.

2.16 Solving systems of distance equations

2.16.1 Principle

We consider here a special occurrence of quadratic equations that describe distances between points in an n dimensional space. Each equation F_k in such system may be written as:

$$\sum_{i=1}^{i=n} (x_i - x_j)^2 + L_k = 0$$

where x_i are unknowns (representing coordinates of points) and x_j may be unknowns of constants. A special occurrence of unknowns are the *virtual points*: the coordinates of these points are linear combination of the coordinates of the points that are defined as the unknowns of the system. To illustrate the concept of virtual points consider 5 fixed points on a rigid body in the 3-dimensional space. The coordinates of any of this point may be expressed as a linear combination of the coordinates of the 4 other points (as soon as these point are not coplanar). The concept of virtual points has been introduced to allow a decrease in the number of unknowns but also because without them distance equations will be redundant and consequently the jacobian of the system of equations will be singular at a solution thereby prohibiting us of using the tests (such as Moore or Kantorovitch) that allows to determine that there is one unique solution in a given box.

Equations involving virtual points may be written as:

$$\sum_{i=1}^{i=n}(\sum \lambda_j x_j - X_k)^2$$

where the x_j are unknowns and the X_k unknowns or constants. Clearly system involving distance equations are of great practical interest and `ALIAS` offers a specific algorithm to deal with such type of systems.

The method proposed in `ALIAS` to solve this type of systems is based on the general procedure using the gradient and hessian. A first difference is that it is not necessary to provide the gradient and hessian function as they are easily derived from the system of equations. Note also that due to the particular structure of the distance equations the interval evaluation leads to exact bounds. Furthermore the algorithm we propose uses a special version of Kantorovitch theorem (i.e. a version that produces a larger ball with a unique solution in it compared to the general version of the theorem), an interval Newton method, a specific version of the simplex method described in section 2.14 and a specific version of the inflation method described in section 3.1.6 (i.e. a method that allows to compute directly the radius of a ball around a solution that will contain only this solution). In addition two simplification rules are used:

- as each function is a sum of square term each of them involving different unknowns we verify if the interval evaluation of the term has a positive part (in the opposite case the current box is discarded) and we may update the unknowns so that the negative part of the term is reduced (this is basically an application of the concept of 2B-consistency). Hence the procedures described in section 2.17 should not be used for distance equations.
- based on the triangle equation: each subset of equations describing the distances between a set of 3 points is detected and the algorithm verify if the triangle equation is satisfied and, in some cases may update the boxes.

The algorithm returns as solution either boxes that satisfy Kantorovitch theorem and therefore are reduced to a point or boxes such that the function evaluations include 0 and have a width lower than a given threshold.

2.16.2 Implementation

A system of m distance functions has a specific description. First consider distance function that involve only points of constant. Such function may be written as:

$$\sum(x_i - x_j)^2 + L_i = 0$$

where the maximal number of square term is n and is described by a row of 2 matrices `APOW`, `ACONS` and a vector `LI`. `APOW` is an integer matrix with $2n$ columns and m rows that contain the unknown number of each term in the function and in which a 0 means that the unknown is a constant. The value of theses constants are given in the matrix of real `ACONS` of size $m \times 2n$. Finally the value of the constant L_i is indicated in the vector `LI` of size m. For example consider a system involving the 4 unknowns x_1, x_2, y_1, y_2 numbered from 1 to 4:

$$x_1^2 + (y_1 - 4)^2 - 6 = 0$$
$$(x_1 - x_2)^2 + (y_1 - y_2)^2 - 12 = 0$$

In that case `APOW`, `ACONS`, `LI` will be:

$$\texttt{APOW} = \begin{pmatrix} 1 & 0 & 3 & 0 \\ 1 & 2 & 3 & 4 \end{pmatrix} \quad \texttt{ACONS} = \begin{pmatrix} 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \texttt{LI} = \begin{pmatrix} \text{-6} \\ \text{-12} \end{pmatrix}$$

The two first elements of the first row of `APOW` (1, 0) describes the first square term of the first equation and state that it is $(x_1 - a)^2$, the value of a being given in the `ACONS`(1,2). Note that each square term must be written as (unknown-unknown or constant)^2 and not as (constant -unknown)^2.

Consider now function involving virtual points. Each square term may thus be written as:

$$(\sum \lambda_j x_j - X_k)^2$$

where X_k may be either a constant, an unknown or the coordinate of a virtual point. Let k be the number of term of the form $\sum \lambda_j x_j$ existing in the system. Such equation is described by a matrix `AVARV` with k rows and a number of columns equal to the number of unknowns. Each term $\sum \lambda_j x_j$ is numbered from 1 to k and the row i of `AVARV` will contain in position j the value of λ_j. The existence of the coordinate of a virtual point in a distance function will be indicated in `APOW` by a negative number whose opposite is the number of the virtual coordinates. Hence if we add to the previous system the third equation:

$$(0.1x_1 - 0.2x_2 - 3)^2 + (0.1y_1 - 0.2y_2 - 1)^2 - 10 = 0$$

the third row of `APOW` will be (-1 0 -2 0), the third row of `ACONS` will be (0 3 0 1) and the third row of `LI` will be -10. `AVARV` will have 2 row and 4 columns and is given as:

$$\texttt{AVARV} = \left(\begin{array}{cccc} 0.1 & \text{-0.2} & 0 & 0 \\ 0 & 0 & 0.1 & \text{-0.2} \end{array} \right)$$

Among the m equations there will be p equations involving virtual points. The system must be written in such way that first are defined the $m - p$ equations not involving virtual points and then the p equations. Note also that in the current implementation inequalities are handled although with less efficiency than equations. The algorithm is implemented as:

```
int Solve_Distance(int DimVar,int DimEq,
                   INTEGER_VECTOR &Type_Eq,
                   INTEGER_MATRIX &APOW,MATRIX &ACONS,VECTOR &LI,
                   int p,int k,MATRIX &AVARV,
                   INTERVAL_VECTOR & TheDomain,
                   int M,
                   double epsilonf,
                   int Stop,
                   INTERVAL_MATRIX & Solution,int Nb,
                   int (* Simp_Proc)(INTERVAL_VECTOR &))
```

The arguments are:

- `DimVar`: number of unknowns
- `DimEq`: number of equations and inequalities in the system
- `Type_Eq`: an integer array of dimension `DimEq` where the j-th element indicates if the j-th function is an equation (value =0), an inequality ≤ 0 (value = -1) or an inequality ≥ 0 (value = 1)
- `p`: number of functions involving virtual points
- `k`: number of virtual points
- `TheDomain`: box in which we are looking for solution of the system
- `M`: the maximum number of boxes which may be stored. In the algorithm we use the reverse storage mode except if the global integer `ALIAS_Parallel_Slave` is set to 1 (its default value is 0)
- `epsilonf`: either the maximal width of the function intervals for a solution if it is not determined by using Newton scheme or the accuracy used in the Newton scheme
- `Stop`: the possible values are 0,1,2

  - 0: the algorithm will look for every solution in `TheDomain`
  - 1: the algorithm will stop as soon as 1 solution has been found
  - 2: the algorithm will stop as soon as `Nb` solutions have been found

- `Solution`: an interval matrix of size (`Nb`,`m`) which will contained the solution intervals. This list is sorted using the order specified by `Order`
- `Nb`: the maximal number of solution which will be returned by the algorithm
- `Simp_Proc`: a user-supplied procedure that take as input the current box and proceed to some further reduction of the width of the box or even determine that there is no solution for this box, in which case it should return -1 (see note 2.3.3). Remember also that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2).

The argument `Simp_Proc` in this procedure may be omitted.

The following variables play also a role in the computation:

- `ALIAS_Diam_Max_Kraw`: if the maximal width of the ranges in a box is lower than this value, then the Krawczyk operator will be used to determine if there is a unique solution in the box (default value: 1.e10)
- `ALIAS_Diam_Max_Newton`: if the maximal width of the ranges in a box is lower than this value, then the interval Newton method will be used either to try to reduce the width of the box or to to ensure that there is no solution of the system in the box (default value: 1.e10)
- `ALIAS_Permute_List`: if the value of this flag is n the algorithm will permute the current list with the largest box in the list of boxes to process (as the algorithm uses systematically the Newton scheme with as initial guess the center of the current box permutation may allow to find quickly new solutions)

2.16.2.1 Return code

This procedure will return:

- $n \geq 0$: number of solutions
- -1: storage space exceeded
- -2: `DimVar` or `DimVar` is a negative number
- -4: an element in `Type_Eq` is not 0, -1 or 1
- $k = -100$: in the mixed bisection mode the number of variables that will be bisected is larger than the number of unknowns
- $k = -200$: one of the value of `ALIAS_Delta3B` or `ALIAS_Max3B` is negative or 0
- $k = -300$: one of the value of `ALIAS_SubEq3B` is not 0 or 1
- $k = -400$: although `ALIAS_SubEq3B` has as size the number of equations none of its components is 1
- $k = -500$: `ALIAS_ND` is different from 0 (i.e. we are dealing with a non-0 dimensional problem, see the corresponding chapter) and the name of the result file has not been specified

Note that in this procedure we use the single bisection mode (we actually select the largest variable as the one that will be bisected) and the reverse storage mode.

2.16.2.2 Inflation and Newton scheme

For the distance equations we use a specific procedure for the inflation method (see section 3.1.6). Indeed in that case it is possible to calculate directly the diameter of the largest ball centered at an approximation of a solution that contains one and only one solution. The procedure is:

```
int ALIAS_Epsilon_Inflation(int Dimension,int Dimension_Eq,
      INTEGER_MATRIX &APOW,MATRIX &ACONS,
      VECTOR &LI,
      int NB_EQV,int NB_VARV,MATRIX &AVARV,
      VECTOR &Amid,
      INTERVAL_VECTOR &P,
      INTERVAL_VECTOR &PP1,
      int type,double epsilon,
      int (* Simp_Proc)(INTERVAL_VECTOR &))
```

We have also a specific implementation of the Newton scheme:

```
int Newton_Fast(int Dimension,int Dimension_Eq,
                   INTEGER_MATRIX &APOW,MATRIX &ACONS,VECTOR &LI,
                   int NB_EQV,int NB_VARV,MATRIX &AVARV,
                   VECTOR &Input,double Accuracy,int Max_Iter,VECTOR &Residu)
```

2.16.2.3 Choosing the right set of equations and variables

For the best efficiency the unknowns has to be well chosen, especially so that the jacobian of the system has full rank at a solution. The use of virtual point should be systematic to avoid having a system that is singular at a solution.

2.16.2.4 Initial domain and simplification procedures

A specific procedure `Bound_Distance` for finding an initial estimate of the search domain has been implemented in the `ALIAS`-Maple package (see the ALIAS-Maple manual): this procedure may provide an initial guess for the solution ranges or improve a given guess.

Note also that the choice of the right coordinate system (for example its origin) and which points should be defined as virtual points may have a large influence on the computation time (see an example in [10]).

2.17 Filtering a system of equation

As mentioned previously the 2B heuristic to improve the solving is to rewrite each equation as the equality of two different terms, to determine if the interval evaluation of both terms are consistent and if not to adjust the interval for one term and by using the inverse function for this term to improve the width for one or more unknowns.

For example imagine that one of the equation is $x^2 - 2x + 1 = 0$. The procedure will introduce a new variable $X = x^2$ such that $X = 2x - 1$ and compute its interval evaluation. If X has a negative upper bound the equation has no solution for the current range for x. If the upper bound U of X is positive then the inverse function of X indicates that x should lie in $[-\sqrt{U}, \sqrt{U}]$: we may then update the interval for x if this is not the case. If the lower bound V of X is positive then the inverse function of X indicates that x should lie outside $[-\sqrt{V}, \sqrt{V}]$. If the range for x is included in this interval, then there is no solution to the equation for this range for x.

We have already seen that the procedure `HullConsistency` of ALIAS-Maple implements this consistency. But this is only a partial implementation as only linear and square terms in the unknowns are considered. A more general implementation has been written by G. Chabert.

To use this implementation you need to prepare first a file such as:

```
size : 1 2
vars : x y
constraints :
x*y^2-x+y-sin(x)-3=0
```

`size` is followed by the number of equations and number of variables, `vars` indicate the unknown names and `constraints` is followed by the equations which are written using a standard mathematical notation.

This description of the system is loaded using the C++ procedure `load_equation`. This procedure has an optional second argument which is a floating point number f whose use will be described later on.

The 2B procedure is then used by calling the C++ procedure `_2b(P)`, where `P` is an interval vector which defines the range for each unknown. If this procedure returns `-1`, then the equations have no solution in `P`, otherwise `P` may have been improved (i.e. one or more of the ranges of `P` may have now a smaller diameter).

The role of f in `load_equation` is that as soon as a range for one (or more) unknown has a diameter that has been decreased by more than f compared to its initial value, then the 2B procedure will be repeated for all equations and all unknowns.

Note:

- the `_2b` procedure allows to deal with a large number of operators but may miss some special cases. If a constraint is not recognized `load_equation` will create a file `_hull_error` with the word `error` in it. Hence if this file is present then you cannot use the `_2b` procedure
- this 2B consistency is implemented in ALIAS-Maple by the procedure `HullConsistencyStrong`
- the `HullConsistency` procedure of ALIAS-Maple may seem to be redundant with `HullConsistencyStrong`. This is not the case as `HullConsistency` implements a simplification of the left hand term of the equation that is not performed by `HullConsistencyStrong`. For example for the equation $x^2 + x * y + y$ `HullConsistencyStrong` will rewrite the equation as $x^2 = -x * y - y$ while `HullConsistency` will use $x^2 = -y * (x - 1)$ which will lead to a better interval evaluation of the left hand term

The `_2b` procedure must not be used with distance equations as the 2B filtering is already implemented in the solver.

Chapter 3

Analyzing systems of equations

3.1 Introduction

The purpose of this chapter is to present some tools enabling to analyze system of equations i.e. to get some a-priori information on the roots without solving the systems. These tools allows one to determine that there is a unique solution in a given box and provides a mean to calculate it in a safe way. Hence they are essential to provide a guaranteed solution for a system of equations.

Note that a generic analyzer based on the `ALIAS` parser has been developed and will be presented in the chapter devoted to the parser. This generic analyzer enable to analyze almost any type of system in which at least one equation is algebraic in at least one of the unknowns.

3.1.1 Moore theorem

3.1.1.1 Mathematical background

Let a system of n equations in n unknowns $F = \{F_i(x_1, \ldots, x_n) = 0, i \in [1, n]\}$ each F_i being at least C^1. Let $\mathbf{X}$ be a range vector for $\{x_1, \ldots, x_n\}$, y a point inside $\mathbf{X}$ and Y an arbitrary nonsingular real matrix. Let define K as:

$$K\mathbf{X} = y - YF(y) + \{I - YF'(\mathbf{X})\}(\mathbf{X} - y)$$

and let r_0 be the norm of the matrix $I - YF'(\mathbf{X})$. If

$$K(\mathbf{X}) \subseteq \mathbf{X} \quad \text{and} \quad r_0 < 1$$

then there is a unique solution [16] of F in $\mathbf{X}$. This unique solution can be found using Krawczyk solving method (see section 2.10).

3.1.1.2 Implementation

The previous test is implemented for $y = Mid(\mathbf{X})$ and $Y = F'^{-1}(y)$. The procedure is implemented as:

```
int Krawczyk_Analyzer(int m,int n,
        INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
        INTERVAL_MATRIX (* J)(int, int, INTERVAL_VECTOR &),INTERVAL_VECTOR &Input)
```

with

- `m`: the number of unknown
- `n`: the number of equations
- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see note 2.3.4.3
- `J`: a function which calculate an interval evaluation of the elements of the jacobian of the equations, see note 2.4.2.2

- `Input`: the ranges for the variables

This procedure returns 1 if there is a unique solution of F in `Input`.

3.1.2 Kantorovitch theorem

3.1.2.1 Mathematical background

Let a system of n equations in n unknowns:

$$F = \{F_i(x_1, \ldots, x_n) = 0, i \in [1, n]\}$$

each F_i being at least C^2. Let $\mathbf{x_0}$ be a point and $U = \{\mathbf{x}/||\mathbf{x}-\mathbf{x_0}|| \leq 2B_0\}$, the norm being $||A|| = \text{Max}_i \sum_j |a_{ij}|$. Assume that $\mathbf{x_0}$ is such that:

1. the Jacobian matrix of the system has an inverse Γ_0 at $\mathbf{x_0}$ such that $||\Gamma_0|| \leq A_0$
2. $||\Gamma_0 F(\mathbf{x_0})|| \leq 2B_0$
3. $\sum_{k=1}^{n} |\frac{\partial^2 F_i(\mathbf{x}}{\partial x_j \partial x_k}| \leq C$ for $i, j = 1, \ldots, n$ and $\mathbf{x} \in U$
4. the constants A_0, B_0, C satisfy $2nA_0B_0C \leq 1$

Then there is an unique solution of $F = 0$ in U and Newton method used with $\mathbf{x_0}$ as estimate of the solution will converge toward this solution [3]. An interesting use of Kantorovitch theorem can be found in section 2.5.

3.1.2.2 Implementation

```
int Kantorovitch(int m,VECTOR (* TheFunction)(VECTOR &),MATRIX (* Gradient)(VECTOR &),
        INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),VECTOR &Input,double *eps)
```

- `m`: number of variables and unknowns
- `TheFunction`: a procedure to compute the value of the equations for given values of the unknowns. This procedure has one arguments which is the value of the unknowns in vector form
- `Gradient`: a procedure to compute the Jacobian matrix of the system in matrix form. This procedure has one arguments which is the value of the unknowns in vector form
- `Hessian`: a procedure to compute the Hessian for the equation for interval value input. This procedure compute the `m X n`, `n` Hessian matrix in interval matrix form.This procedure has 3 arguments `l1,l2,X`. The function should return the value of the Hessian of the equations from `l1` to `l2` The Hessian of the first equation is stored in hess(1..`n`,1...`n`), the Hessian of the second equation in hess(`n`+1..2`n`,1..`n`) and so on
- `Input`: the value of the variables which constitute the center of the convergence ball

Another implementation, consistent with the procedure used in the general solving algorithm (see section 2.5) is:

```
int Kantorovitch(int m,
        INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
        INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
        INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &), VECTOR &Input,double *eps)
```

There is also an implementation of Kantorovitch theorem for univariate polynomial, see section 5.2.12.

3.1.2.3 Return code

This procedure return an integer k:

- k= -1: the Jacobian matrix has no inverse at `Input`
- k=0: the procedure has failed to determine a convergence ball centered at `Input`
- k=1: the procedure has found that an unique solution exist within the interval [`Input-eps,Input+eps`]

3.1.3 Rouche theorem

3.1.3.1 Mathematical background

Let a system of n equations in n unknowns:

$$F = \{F_i(x_1, \ldots, x_n) = 0, i \in [1, n]\}$$

We will denote by $F^{(k)}$ the matrix of the derivatives of order k of F with respect to the variable. Let us consider a point X_0 and define γ as

$$\gamma = Sup(\frac{||(F^{(1)}(X_0))^{-1}F^{(k)}(X_0)||^{1/k-1}}{k!}) \quad k \geq 2$$

$$\beta = ||(F^{(1)}(X_0))^{-1}F(X_0)||$$

and $\alpha = \beta\gamma$. If α is strictly lower than $3 - 2\sqrt{2}$, then F has a single root in a ball centered at X_0 with radius $(1 + \alpha - \sqrt{\alpha^2 - 6 * \alpha + 1})/(4 * \gamma)$ and the Newton scheme with initial guess X_0 will converge to the solution.

The most difficult part for using this theorem is to determine γ. For algebraic equations it is easy to determine a value k_1, that we will call the *order* of Rouche theorem, such that $F^{(k_1)} = 0$ and consequently γ may be obtained by computing

$$S_k = \frac{||(F^{(1)}(X_0))^{-1}F^{(k)}(X_0)||^{1/k-1}}{k!}$$

for all k in $[2, k_1 - 1]$ and taking γ as the Sup of all S_k.

For non algebraic finding γ requires an analysis of the system.

Rouche theorem may be more efficient than Moore or Kantorovitvh theorems. For example when combined with a polynomial deflation (see section 5.9.6) it allows one to solve Wilkinson polynomial of order up to 18 with the C++ arithmetic on a PC, while stand solving procedure fails for order 13.

3.1.3.2 Implementation

Rouche theorem is implemented in the following way:

- Rouche theorem is checked with respect to the mid-point of a box
- if Roucche theorem is satisfied, then a limited number of Newton iteration is performed to check if Newton indeed converge. If this is the case a ball that include a single solution has been determined
- if a ball has been determined, then, optionaly an inflation procedure)see section 3.1.6) is used to try to enlarge the ball

The syntax of the procedure is:

```
int Rouche(int DimensionEq,int DimVar,int order,
           INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
           INTERVAL_VECTOR (* Jacobian)(int, int, INTERVAL_VECTOR &),
           INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
           INTERVAL_VECTOR (* OtherDerivatives)(int, int, INTERVAL_VECTOR &),
           double Accuracy,
           int MaxIter,
           INTERVAL_VECTOR &Input,
           INTERVAL_VECTOR &UnicityBox)
```

where

- `DimensionEq`: number of equations
- `DimVar`: number of variables
- `order`: the order for Rouche theorem minus 1
- `TheIntervalFunction`: a procedure in `MakeF` format for computing an interval evaluation of the equations
- `Jacobian`: a procedure in `MakeF` format that computes the jacobian row by row
- `Gradient`: a procedure that compute the jacobian in `MakeJ` format
- `OtherDerivatives`: a procedure in `MakeF` format that computes the derivative of order larger or equal to 2, row by row. This procedure returns an interval vector of dimension `orderDimensionEq`^{2}

The parameters `Accuracy` is used in the Newton scheme to determine if Newton has converged i.e. if the residues are lower than `Accuracy`. A maximum of `MaxIter` iterations are performed. The solution found with Newton is stored in `ALIAS_Simp_Sol_Newton_Numerique` while a copy of the unicity box is available in `ALIAS_Simp_Sol_Newton`

If a ball with a single solution has been found it will be returned in `UnicityBox` and the procedure returns 1, otherwise it returns 0.

If the flag `ALIAS_Always_Use_Inflation` is set to 1, then an inflation procedure is used to try to enlarge the box up to the accuracy `ALIAS_Eps_Inflation`.

3.1.4 Interval Newton

The classical interval Newton method is embedded in the procedure `GradientSolve` and `HessianSolve` but may also be useful in other procedures. Furthermore this method relies on the use of the product $J^{-1}(X_0)J(X)$ where J is the Jacobian of the system of equations and $J^{-1}(X_0)$ the inverse of J computed at some particular point X_0. In the classical method this product is cimputed numerically and this does not take into account that the element of J are functions of the same parameters. For example if the first column of J is (x, x) where x is some parameter with interval value, the first element of $J^{-1}(X_0)J(X)$ will be computer as

$$a_{11}x + a_{12}x$$

where $(a_{11}, a_{12}$ are the elements of $J^{-1}(X_0)$. Clearly the double occurence of x in the numerical evaluation of the elements may lead to an overestimation of the elements: this element should be written as $x(a_{11} + a_{12})$ which is optimal in term of interval evaluation. Furthermore it may also be interesting to have the derivatives of each element of the product in order to improve the interval evalation of the matrix product. Indeed the interval evaluation of $J^{-1}(X_0)J(X)$ plays a very important role in the interval Newton method either for filtering a box for possible solution or for determining that a box includes a solution of the system.

The procedure `IntervalNewton` is a sophisticated interval Newton algorithm that allows one to introduce knowledge on the product $K = J^{-1}(X_0)J(X)$ in the classical scheme. Its syntax is:

```
int IntervalNewton(int Dim,INTERVAL_VECTOR &P,INTERVAL_VECTOR &FDIM,
                   INTERVAL_MATRIX &Grad,MATRIX &GradMid,
                   MATRIX &InvGradMid,
                   int hasBGrad,
                   INTERVAL_VECTOR (* BgradFunc)(int,int,INTERVAL_VECTOR &),
                   INTERVAL_MATRIX (* BgradJFunc)(int, int,INTERVAL_VECTOR &),
                   int grad1,
                   int grad3B1)
```

where

- `Dim`: the size of the system
- `P`: an interval vector that describes the range for the unknowns

- FDIM: interval value of the equation at the mid-point of P
- Grad: interval jacobian at P
- GradMid: jacobian at the mid-point of P
- hasBgrad: a flag that indicates how K will be calculated:
  - 0: K will be calculated numerically
  - 1: K will be calculated using the procedure BgradFunc
  - 2: K will be calculated using the procedure BgradFunc and the derivatives of the elements of K available through the procedure BgradJFunc
- BgradFunc: a user-provided procedure in MakeF format that calculate the element of K, row by row
- BgradJFunc: a user-provided procedure in MakeJ format that calculate the derivatives of the elements of K
- grad1: if hasBgrad is 1 or 2 we use the procedure BgradFunc to evaluate K when we are in the 3B filter if grad1 is set to 1. If set to 0 we use BgradFunc only when dealing with the full box
- grad3B1: if 1 we use the procedure that evaluates K through BgradFunc even if we are in the 3B case. If 2 we use both BgradFunc and BgradJFunc

The procedure returns -1 if no solution of the system exists in P, 1 if P has been improved, 0 otherwise. Note that the procedure BgradFunc and BgradJFunc may require the availability of the mid-matrix GradMid: therefore a global MATRIX should be made available, initialized with GradMid.

Various variants of IntervalNewton are available:

```
int IntervalNewton(int Dim,INTERVAL_VECTOR &P,INTERVAL_VECTOR &FMID,
                  INTERVAL_MATRIX &Grad,MATRIX &GradMid,MATRIX &InvGradMid)
```

is the classical interval Newton method with hasBgrad=grad1=grad3B1=0.

```
int IntervalNewton(int Dim,INTERVAL_VECTOR &P,int DimVar,int DimEq,
          int TypeGradMid,MATRIX &InvGradMid,
          INTERVAL_VECTOR (*TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &))
```

is also the classical interval Newton method for a system having DimVar unknowns and DimEq equations (here DimVar and DimEq are not required to have the same value: only the Dim first equations will be considered). The flag TypeGradMid is used to determine how the mid jacobian matrix is calculated: if 0 this matrix is calculated for the mid-point of P, if 1 the mid-jacobian is calculated as the mid-matrix of the interval jacobian calculated for P.

```
int IntervalNewton(int Dim,INTERVAL_VECTOR &P,int DimEq,int DimVar,
                   int has_BGrad,
                   INTERVAL_VECTOR (* BgradFunc)(int,int,INTERVAL_VECTOR &),
                   INTERVAL_MATRIX (* BgradJFunc)(int, int,INTERVAL_VECTOR &),
                   int grad1,int grad3B1,
                   int TypeGradMid,
                   MATRIX &GradFuncMid,
                   MATRIX &InvGradFuncMid,
                   INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
                   INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &))
```

Here the the mid jacobian GradFuncMid and its inverse InvGradFuncMid will be provided by the procedure.

3.1.5 Miranda theorem

Miranda theorem provides a simple way to determine if there is one, or more, solution of a system of equations in a given box. It has the advantage of not requiring the derivatives of the equations but the drawback of not provinding the proof of the existence of a single solution in the box.

3.1.5.1 Mathematical background

Let a system $F(X) = 0$ with $X = \{x_1, \ldots, x_n\}$. Let us consider a ball $\mathcal{B} = \{[\underline{x_1}, \overline{x_1}], \ldots, [\underline{x_n}, \overline{x_n}]$ for X and define

$$[X]_j^+ = \{X \in \mathcal{B} \text{ such that } x_j = \overline{x_j}\}$$
$$[X]_j^- = \{X \in \mathcal{B} \text{ such that } x_j = \underline{x_j}\}$$

for j in $[1, n]$. If

$$F_j(X) \geq 0 \text{and } F_j(Y) \leq 0 \quad \forall\ X \in [X]_j^+, \forall\ Y \in [X]_j^-, j = 1, \ldots, n$$

or if

$$F_j(X) \leq 0 \text{and } F_j(Y) \geq 0 \quad \forall\ X \in [X]_j^+, \forall\ Y \in [X]_j^-, j = 1, \ldots, n$$

then F has at least one zero in $\mathcal{B}$ [15].

3.1.5.2 Implementation

The simplest implementation of the Miranda theorem is

```
int Miranda(int Dim,INTERVAL_VECTOR (* F)(int,int,INTERVAL_VECTOR &),
           INTERVAL_VECTOR &Input)
```

where `Dim` is the number of equations, `Input` is a ball for the variables and `F` is a procedure in `MakeF` format that allows to compute an interval evaluation of the equations. This procedure returns 1 if Miranda theorem is satisfied for `Input`, 0 otherwise. This implementation is embedded in the `Solve_General_Interval` solving algorithm.

Another implementation uses the derivatives for improving the interval evaluation:

```
\begin{verbatim}
int Miranda(int Dim,
           INTERVAL_VECTOR (* F)(int,int,INTERVAL_VECTOR &),
           INTERVAL_MATRIX (* J)(int,int,INTERVAL_VECTOR &),
           INTERVAL_VECTOR &Input)
```

`J` is a procedure in `MakeJ` format that allows to compute the derivative of the equations.

3.1.6 Inflation

3.1.6.1 Mathematical background

Let a system $F(X) = 0$, J the Jacobian matrix of this system and $X0$ a solution of the system. The purpose of the inflation method is to build a box that will contain only this solution. Let $B(X0)$ be a ball centered at $X0$: if for any point in B J is not singular, then the ball contains only one solution of the system.

The problem now is to determine a ball such for any point in the ball the Jacobian is regular. Let H be the matrix $J(X0)^{-1}J(B)$ whose components are intervals. Let u be the diagonal element of H having the lowest absolute value, let v_i be the maximum of the absolute value of the sum of the elements at row i of H, discarding the diagonal element of the row and let v be the maximum of the v_i's. If $u > v$, then the matrix is denoted *diagonally dominant* and all the matrices $J(B)$ are regular [19].

Let ϵ be a small constant: we will build incrementally the ball B by using an iterative scheme defined as:

$$B_0 = X0$$
$$B_n = [B_{n-1} - \epsilon, B_{n-1} + \epsilon]$$

that will be repeated until B_n is no more diagonally dominant. ϵ Note that in some cases (see for example section 2.16) it is possible to calculate directly the largest possible ϵ so that all the matrices in $J([B_0 - \epsilon, B_0 + \epsilon])$ are regular without relying on the iterative scheme.

3.1.6.2 Implementation

```
int ALIAS_Epsilon_Inflation(int Dimension,int Dimension_Eq,
       INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),
       VECTOR &X0,
       INTERVAL_VECTOR &B)
```

Note that the `Hessian` argument is not used in this procedure. This routine will return 1 if the inflation has succeeded. The value of ϵ can be found in the global variable. `ALIAS_Eps_Inflation`

Chapter 4

Analyzing trigonometric equations

4.1 Introduction

The purpose of this chapter is to describe tools which can be used to analyze either one trigonometric equation in one variable or a system of trigonometric equations. These tools may be used either to improve the efficiency of the solving algorithm described in the previous chapter or even to avoid calling them if some constraints are imposed on the location of the solution.

4.2 Number of roots of trigonometric equation

The purpose of this section is to present an algorithm which enable to determine how many roots has an equation F in the unknown x of the form:

$$F = \sum a_k \sin^m(x) \cos^n(x) \tag{4.1}$$

with m in $[0, M]$ and n in $[0, N]$, both m, n being integers, and x in $(\underline{x}, \overline{x})$.

4.2.1 Mathematical background

We use the half angle tangent substitution. If θ is the unknown we define T as:

$$T = \tan(\frac{\theta}{2})$$

Then we have:

$$\sin(\theta) = \frac{2T}{1+T^2} \qquad \cos(\theta) = \frac{1-T^2}{1+T^2}$$

Note that the change of variable is not valid if $\theta = \pm\pi$. In that case it will be preferable to define $\alpha = \theta + \pi$ and to transform the initial into an equation in α. Then the change of variable may be applied.

Using the above relation any trigonometric equation can be transformed into a polynomial equation which is analyzed using the tools of chapter 5.

It remains to define an interval for angles that we will denote an *angle interval*. The element of an angle interval is usually defined between 0 and 2π (although in most of the following procedures any value can be used when not specified: internally the element of the angle interval are converted into value within this range). A difference between numbers interval (`INTERVAL`) and angle interval is that the lower bound of an angle interval may be larger than the upper bound. Indeed the order in an angle interval is important: for example the angle intervals $[0,\pi/4]$ and $[\pi/4,0]$ are not the same.

4.2.2 Implementation

The purpose of the implementation is first to convert the trigonometric equation in sine and cosine of θ:

$$F = \sum a_k \sin^m(\theta) \cos^n(\theta) \tag{4.2}$$

into a polynomial in either $\tan(\theta/2)$ or $\tan((\pi+\theta)/2)$. A first procedure return an upper bound of the degree of the resulting polynomial:

```
int Degree_Max_Convert_Trigo_Interval(int n,VECTOR &A,INTEGER_VECTOR &SSin,INTEGER_VECTOR &CCos);
int Degree_Max_Convert_Trigo_Interval(int n,INTEGER_VECTOR &A,
                         INTEGER_VECTOR &SSin,INTEGER_VECTOR &CCos);
```

with:

- `n`: number of terms of the equation
- `A`: the coefficients of each term which may be either real or integer
- `Ssin`: the sine power of each term
- `Ccos`: the cosine power of each term

Then we may use a procedure which compute the coefficients of the polynomial equation using the following procedures:

```
VOID Convert_Trigo_Interval(int n,VECTOR &A,INTEGER_VECTOR &SSin,
                         INTEGER_VECTOR &CCos,VECTOR &Coeff,int *degree);
VOID Convert_Trigo_Interval(int n,INTEGER_VECTOR &A,INTEGER_VECTOR &SSin,
                         INTEGER_VECTOR &CCos,INTEGER_VECTOR &Coeff,int *degree);
VOID Convert_Trigo_Interval(int n,INTERVAL_VECTOR &A,INTEGER_VECTOR &SSin,
                         INTEGER_VECTOR &CCos,INTERVAL_VECTOR &Coeff,int *degree);
```

with:

- `n`: number of terms of the equation
- `A`: the coefficients of each term which may be either real or integer
- `Ssin`: the sine power of each term
- `Ccos`: the cosine power of each term
- `Coeff`: the coefficients of the polynomial
- `degree`: the final degree of the polynomial

The previous procedures use the substitution $T=\tan(\theta/2)$ which is not valid for $\theta=\pi$. In that case we may use instead the substitution $T=\tan((\pi+\theta)/2)$ and the coefficient of the resulting polynomial may be determined using the following procedures:

```
VOID Convert_Trigo_Pi_Interval(int n,VECTOR &A,INTEGER_VECTOR &SSin,
            INTEGER_VECTOR &CCos,INTEGER_VECTOR &Coeff,int *degree);
VOID Convert_Trigo_Pi_Interval(int n,INTEGER_VECTOR &A,INTEGER_VECTOR &SSin,
            INTEGER_VECTOR &CCos,INTEGER_VECTOR &Coeff,int *degree);
VOID Convert_Trigo_Pi_Interval(int n,INTERVAL_VECTOR &A,INTEGER_VECTOR &SSin,
            INTEGER_VECTOR &CCos,INTEGER_VECTOR &Coeff,int *degree);
```

Similar procedures exists for interval trigonometric equations i.e. equations where the coefficients `A` are intervals. In that case `degree` will no more an integer but an `INTERVAL` which indicate the lowest and highest degree of the resulting polynomial. In some case the number of roots of the trigonometric equation may exceed the degree of the equivalent polynomial. For example the equation $\sin\theta=0$ has the roots $0,\pi$ while the degree of the equivalent polynomial is only 1. In all cases the total number of roots of the trigonometric equation will never exceed the degree+2.

Having determined the equivalent polynomial you my use the tools described in section 5 for determining the number of roots of the trigonometric equation. But you still have to manage the search interval. The following procedure is able to determine this search interval and to determine the number of roots of the trigonometric equation:

```
int Nb_Root_Trigo_Interval(int n,VECTOR &A,INTEGER_VECTOR &SSin,
                           INTEGER_VECTOR &CCos,REAL Inf,REAL Sup)
```

with:

- `n`: number of terms of the equation
- `A`: the coefficients of each term which may be either real or integer
- `Ssin`: the sine power of each term
- `Ccos`: the cosine power of each term
- `Inf`: the lower bound of the search interval
- `Sup`: the upper bound of the search interval

On success this procedure returns a number greater or equal to 0 and returns -1 if it has failed. Failure occurs either if the equation is equal to 0 or if Sturm method failed to determine the number of roots of the equivalent polynomial equation (this will happen if `Inf` or `Sup` are exact root of the equation).

4.2.3 Example

Consider the equation:

$$(\sin(t) - 1/2)\left(\sin(t) - 1/2\sqrt{3}\right)\left(\cos(t) - 1/2\sqrt{2}\right) = 0$$

which has as roots:$\pi/6(0.5235)$,$\pi/4(0.7853)$, $\pi/3(1.047)$,$2\pi/3(2.094)$,$5\pi/6(2.617)$,$7\pi/4(5.497)$. This equation is equivalent to:

$$\sin^2(t)\cos(t) - 0.707106781\sin^2(t) - 1.366025404\sin(t)\cos(t) + 0.9659258263\sin(t) + 0.4330127\cos(t) - 0.3061862179$$

The `A, SSin, CCos` vectors have the following values:

A	SSin	CCos
1	2	1
-0.7071067810	2	0
-1.366025404	1	1
0.9659258263	1	0
0.4330127020	0	1
-0.3061862179	0	0

If `Inf`=0 and `Sup`=0.8 the procedure indicates that there are two roots corresponding to $\pi/6 = 0.5235987758$ and $\pi/4 = 0.7853981635$.

4.3 Bound on the roots of trigonometric equation

The purpose of this section is to present an algorithm which enable to determine bounds on the roots of an equation F in the unknown x of the form:

$$F = \sum a_k \sin^m(x)\cos^n(x) \tag{4.3}$$

with m in $[0, M]$ and n in $[0, N]$ and x in $(\underline{x}, \overline{x})$.

4.3.1 Implementation

This procedure return angle intervals included in the range $[0, 2\pi]$ which contain the roots of the trigonometric equation.

```
int Bound_Root_Trigo_Interval(int n,VECTOR &A,INTEGER_VECTOR &SSin,
            INTEGER_VECTOR &CCos,int *nbsol,VECTOR &Inf,VECTOR &Sup);
```

with:

- `n`: number of terms of the equation
- `A`: the coefficients of each term which may be either real or integer
- `Ssin`: the sine power of each term
- `Ccos`: the cosine power of each term
- `nbsol`: the number of angle intervals which are returned by the procedure (at most 4)
- `Inf`: the lower bound of the returned angle interval
- `Sup`: the upper bound of the returned angle interval

Note that the size of `Inf, Sup` should be 4 at least. In case of failure the procedure return `-1`, 1 on success.

4.3.1.1 Example

Consider the equation:

$$(\sin(t) - 1/2)\left(\sin(t) - 1/2\sqrt{3}\right)\left(\cos(t) - 1/2\sqrt{2}\right) = 0$$

which has as roots:$\pi/6(0.5235)$,$\pi/4(0.7853)$, $\pi/3(1.047)$,$2\pi/3(2.094)$,$5\pi/6(2.617)$,$7\pi/4(5.497)$. The procedure returns 2 angle intervals: [0.517117,2.62532], [5.49664,5.50734].

4.4 Utilities for trigonometric equation

4.4.1 Inclusion in an angle interval

The procedure:

```
int Angle_Ok_Interval(double angle,double b1,double b2);
```

return 1 if the angle `angle` belongs to the angle interval [`b1`,`b2`], 0 otherwise. The angle `angle,b1, b2` should have a value within [0,2π].

4.4.2 Distance between two angles

The procedure:

```
double Distance_Angle(double a1,double a2)
```

return the smallest distance in radian between the two angles `a1,a2`.

4.4.3 Generalized inverse trigonometric functions

Assume that we have $\cos(\beta) = U$ where U is an interval and β should lie in an arbitrary range. The procedure `Filtre_Arc_Cos` allows to update the range for β. It returns -1 is β and U are incompatible, 0 otherwise. Its syntax is:

```
int Filtre_Arc_Cos(INTERVAL &U,INTERVAL &beta)
```

A similar procedure exist for the inverse sine with

The procedure `Arc_Cos_Multiple` allows to determine all possible ranges for β, assuming that β is restricted to a range included in $[-2\pi, 2\pi]$::

```
int Arc_Cos_Multiple(INTERVAL &U,INTERVAL &beta,INTERVAL_VECTOR &BETA)
```

This procedure returns the number of possible ranges for β and their values in `BETA`.

Chapter 5

Analyzing univariate polynomials

5.1 Introduction

In this chapter we intend to determine some information on the roots of an univariate polynomial without solving it. A polynomial is defined by the list of its coefficient ordered along increasing power in a data structure of type `VECTOR`. For most of the procedures defined in the sequel the coefficients may also intervals, in which case the data structure is of type `INTERVAL_VECTOR` (this type of polynomial will be called *interval polynomial*).

5.2 Finding bounds on the roots

5.2.1 First Cauchy theorem

5.2.1.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + ...a_0 = 0$$

with $a_0 a_n \neq 0$. Let

$$A = max\{\mid a_{n-1} \mid, \mid a_{n-2} \mid, ..., \mid a_0 \mid\} \quad A' = max\{\mid a_n \mid, \mid a_{n-1} \mid, ..., \mid a_1 \mid\}$$

Then the modulus of any root x_k of $P(x) = 0$ verify:

$$\frac{1}{1 + \frac{A'}{|a_0|}} < \mid x_k \mid < 1 + \frac{A}{\mid a_n \mid}$$

5.2.1.2 Implementation

This procedure is implemented as:

```
int Cauchy_First_Bound_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `Bound`: the interval on the absolute value of the roots

This procedure returns 0 if `Degree`=0, `Coeff(1)`=0 or `Coeff(Degree+1)`=0 and `Degree`=1. On success the return code is 1. There is also an implementation for interval polynomial:

```
int Cauchy_First_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Bound);
```

This procedure returns an interval ($\underline{Bound}, \overline{Bound}$) such that for any polynomial in the set and for any root x of this polynomial $\underline{Bound} \leq |x| \leq \overline{Bound}$. A failure code of 0 is returned if `Coeff(1)` or `Coeff(Degree+1)` contains 0 or if `Degree`=0. On success the returned code is 1.

5.2.1.3 Example

Let $P = x^3 - x^2 + 2x - 3$ and the procedure:

```
Coeff(1)= -3;Coeff(2)=2;Coeff(3)=-1;Coeff(4)=1;
Num=Cauchy_First_Bound_Interval(3,Coeff,Bound);

Coeff_App(1)= INTERVAL(-3.1,-2.9);Coeff_App(2)=INTERVAL(1.9,2.1);
Coeff_App(3)=INTERVAL(-1.1,-0.9);Coeff_App(4)=INTERVAL(0.9,1.1);
Num=Cauchy_First_Bound_Interval(3,Coeff_App,Bound);
```

In the first case the procedure find that the absolute value of the roots lie in [0.6,4] while in the second case the range is [0.58,4.44444].

5.2.2 Second Cauchy theorem

5.2.2.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = x^n + a_1x^{n-1} + ...a_n = 0$$

with $a_n \neq 0$. Let $a_{m_1}, a_{m_2}, \ldots$ with $m_1 > m_2 > \ldots$ the k strictly negative coefficients of P. Then all the positive real roots of P verify [13]:

$$x \leq \text{Max}\{(k|a_{m_1}|)^{1/m_1}, (k|a_{m_2}|)^{1/m_2}, \ldots\}$$

Note that if $k = 0$ all the roots are negative according to Descartes Lemma (see section 5.5.1).

5.2.2.2 Implementation

This procedure is implemented as:

```
int Cauchy_Second_Bound_Interval(int Degree,VECTOR &Coeff,double *Bound);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `bound`: the upper bound of the positive roots

This procedure returns 0 if the method cannot be applied because `Degree`=0 or `Degree`=1 and `Coeff(2)`=0. On success the return code is 1. It is also possible to determine the lower bound of the positive roots using:

```
int Cauchy_Second_Bound_Inverse_Interval(int Degree,VECTOR &Coeff,double *Bound);
```

This procedure fail and returns 0 if `Degree`=0, `Degree`=1 and `Coeff(2)`=0 or `Coeff(1)`=0. There is also an implementation for interval polynomial:

```
int Cauchy_Second_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL Bound);
```

This procedure will return a failure code of 0 if `Degree`=0, `Degree`=1 and `Coeff(2)` contain 0, or if `Coeff(Degree+1)` contains 0. In that case if `Bound`=[a,b], then for all polynomials in the set the positive roots are all lower than b while for some polynomial in the set the roots are lower than a. Equivalently we have:

```
int Cauchy_Second_Bound_Inverse_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL Bound);
```

In that case if `Bound`=[a,b], then for all polynomials in the set the positive roots are all lower than a while for some polynomial in the set the roots are lower than b.

A global procedure enable to get at the same time the upper and lower bound of the positive roots:

```
int Cauchy_Second_Bound_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
int Cauchy_Second_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,
                            INTERVAL &Lower,INTERVAL &Upper);
```

In the latter case the interval lower bound is in `Lower` and the interval upper bound in `Upper`. It is also possible to determine the lower bound of the negative roots using the procedures:

```
int Cauchy_Second_Bound_Negative_Interval(int Degree,VECTOR &Coeff,double *Bound);
int Cauchy_Second_Bound_Negative_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Bound);
```

while the upper bound of the negative roots can be determined using:

```
int Cauchy_Second_Bound_Negative_Inverse_Interval(int Degree,VECTOR &Coeff,double *Bound);
int Cauchy_Second_Bound_Negative_Inverse_Interval(int
Degree,INTERVAL_VECTOR &Coeff,
                  INTERVAL &Bound);
```

Both the upper and lower bound of the negative roots can be found using:

```
int Cauchy_Second_Bound_Negative_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
int Cauchy_Second_Bound_Negative_Interval(int Degree,INTERVAL_VECTOR &Coeff,
                                  INTERVAL &Lower,INTERVAL &Upper);
```

5.2.2.3 Example

Let $P = x^3 - x^2 + 2x - 3$ and the procedure:

```
Coeff(1)= -3;Coeff(2)=2;Coeff(3)=-1;Coeff(4)=1;
Num=Cauchy_Second_Bound_Interval(3,Coeff,&bound);
Coeff_App(1)= INTERVAL(-3.1,-2.9);Coeff_App(2)=INTERVAL(1.9,2.1);
Coeff_App(3)=INTERVAL(-1.1,-0.9);Coeff_App(4)=INTERVAL(0.9,1.1);
Num=Cauchy_Second_Bound_Interval(3,Coeff_App,Bound);
```

In the first case we find that all the roots are lower than 2 and in the second that the roots have bounds [2.44444,2.44444]. If we have used the lower bound procedures we will have found that all the roots are positive.

5.2.3 Third Cauchy theorem

5.2.3.1 Mathematical background

This procedure calculates a bound for the absolute value of the roots of the polynomial, see Handbook of numerical analysis, Ciarlet, Lions, Volume 3.

5.2.3.2 Implementation

```
int Cauchy_Third_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Bound)
```

`Coeff[1]` cannot be 0. The procedure:

```
int Cauchy_All_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,
                             INTERVAL &Bound)
```

will return the result of a successive application of the first and third Cauchy bounds.

5.2.4 Lagrange-MacLaurin theorem

5.2.4.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = a_0x^n + a_1x^{n-1} + ...a_n = 0$$

Assume $a_0 > 0$ and let a_k $(k \geq 1)$ be the first negative coefficients of $P(x)$ (if $P(x)$ has no negative coefficients then there is no positive real root).

The upper bound M of the value of the positive real root is:

$$M = 1 + \sqrt[k]{\frac{B}{a_0}}$$

where B is the greatest absolute value of the negative coefficients of $P(x)$,[3],[13].

If we define:

$$P(y) = a_n y^n + a_{n-1} y^{n-1} + ... + a_0$$

Then the upper bound of the positive real roots of $P(y)$ is the lower bound m of the positive real root of $P(x)$. Consequently if k and B are computed for the polynomial $P(y)$ then

$$m = \frac{1}{1 + \sqrt[n-k]{\frac{B}{a_n}}}$$

5.2.4.2 Implementation

This procedures is implemented as:

```
int MacLaurin_Bound_Interval(int Degree,VECTOR &Coeff,double *Bound);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `bound`: the upper bound of the positive roots

This procedure fail and returns 0 if `Degree`=0, `Degree`=1 and `Coeff(2)`=0 and if `Coeff(Degree+1)`=0.

On success the return code is 1. There is also a procedure for the interval polynomial:

```
int MacLaurin_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL Bound);
```

This procedure fail and returns 0 if `Degree`=0, `Degree`=1 and 0 ∈`Coeff(2)` and if 0 ∈`Coeff(Degree+1)`. In that case if `Bound`=[a,b], then for all polynomials in the set the roots are all lower than b while for some polynomial in the set the roots are lower than a.

It is also possible to determine the lower bound of the positive roots using the procedures:

```
int MacLaurin_Bound_Inverse_Interval(int Degree,VECTOR &Coeff,double *Bound);
int MacLaurin_Bound_Inverse_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Bound);
```

In the later case if `Bound`=[a,b], then for all polynomials in the set the roots are all lower than a while for some polynomial in the set the roots are lower than b. To get directly both lower and upper bound of the positive roots you may use:

```
int MacLaurin_Bound_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound)
int MacLaurin_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Lower, INTERVAL &Upper)
```

In the latter case the interval lower bound is in `Lower` and the interval upper bound in `Upper`.

It is also possible to determine the lower bound of the negative roots using the procedures:

```
int MacLaurin_Bound_Negative_Interval(int Degree,VECTOR &Coeff,double *Bound);
int MacLaurin_Bound_Negative_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Bound);
```

Similarly it is possible to determine the upper bound of the negative roots using the procedures:

```
int MacLaurin_Bound_Negative_Inverse_Interval(int Degree,VECTOR &Coeff,double *Bound);
int MacLaurin_Bound_Negative_Inverse_Interval(int Degree,
                              INTERVAL_VECTOR &Coeff,INTERVAL &Bound);
```

To get directly both lower and upper bound of the negative roots you may use:

```
int MacLaurin_Bound_Negative_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound)
int MacLaurin_Bound_Negative_Interval(int Degree,INTERVAL_VECTOR &Coeff,
                                  INTERVAL &Lower, INTERVAL &Upper)
```

5.2.4.3 Example

Let $P = x^3 - x^2 + 2x - 3$ and the procedure:

```
Coeff(1)= -3;Coeff(2)=2;Coeff(3)=-1;Coeff(4)=1;
Num=MacLaurin_Bound_Interval(3,Coeff,&bound);
Coeff_App(1)= INTERVAL(-3.1,-2.9);Coeff_App(2)=INTERVAL(1.9,2.1);
Coeff_App(3)=INTERVAL(-1.1,-0.9);Coeff_App(4)=INTERVAL(0.9,1.1);
Num=MacLaurin_Bound_Interval(3,Coeff_App,Bound);
```

In the first case we find that all the roots are lower than 4 and in the second that the roots have bounds [4.44444,4.44444]. If we have used the lower bound procedures we will have found that all the roots are greater than -1.

5.2.5 Laguerre method

5.2.5.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = a_0x^n + a_1x^{n-1} + ...a_n = 0$$

with $a_0a_n \neq 0$, $a_0 > 0$. Let's define the sequence :

$$f_0 = a_0, f_1 = xf_0 + a_1, \ldots, f_i = xf_{i-1} + a_i, \ldots, f_n = f_{n-1} + a_n = P$$

If it exists a real c such that $f_i(c) \geq 0$ for all i in [0,n], then $P(x) > 0$ for all $x > c$. Consequently all the roots of P are lower than c [13]. To find c the following scheme can be used:

1. let c be such that $f_1(c) \geq 0$
2. let k the smallest integer such that either $k = n + 1$ or $k \leq n$ and $f_k(c) \leq 0$
3. if $k \leq n$ then substitute c by c' such that $f_k(c') \geq 0$ and go to 2
4. return c

A consequence of Laguerre theorem is that the best bound cannot be lower than $-a_1/a_0$.

5.2.5.2 Implementation

This procedure is able to give upper and lower bound on the value of the roots. In the implementation we define c as the smallest real which satisfy $f_1(c) \geq 0$. If we find k such that $f_k(c) \leq 0$, then we increase c by a given positive value `sens` and start again. We limit the number of iteration of the scheme by giving a maximal value for the number of iteration:

```
int Laguerre_Bound_Interval(int Degree,VECTOR &Coeff,double sens,int MaxIter,double *bound);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `sens`: a positive real indicating the increase of c in the scheme
- `MaxIter`: the maximum number of iteration. If this number is exceeded the procedure returns 0.

- `bound`: the upper value of the real root

This procedure fail and returns 0 if `Degree`=0, `Coeff(1)`=0, if `Coeff(Degree+1)`=0 and if the number of iteration exceed `MaxIter`. On success the return code is 1. Note that the bound given by Laguerre (assuming that `Coeff(Degree+1)` is positive) cannot be lower than `-Coeff(Degree)/Coeff(Degree+1)`.

The lower bound of the root may be determined by:

```
int Laguerre_Bound_Inverse_Interval(int Degree,VECTOR &Coeff1,double amp_sens,
                      int MaxIter,double *bound);
```

We have also a procedure which determine upper and lower bound for the real roots:

```
int Laguerre_Bound_Interval(int Degree,VECTOR &Coeff,double sens,int MaxIter,INTERVAL &Bound);
```

All the real roots lie within `Bound`. We may also use this procedure for interval polynomial:

```
int Laguerre_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,double sens,
                              int MaxIter,INTERVAL &Bound);
```

This procedure fail and returns 0 if `Degree`=0, $0 \in$ `Coeff(1)`, if $0 \in$`Coeff(Degree+1)` and if the number of iteration exceed `MaxIter`. If `Bound`=[a,b], then the real roots of all the polynomial in the set are lower than b and for some polynomial in the set the roots may be lower than a. A similar procedure exists for upper and lower bound.

```
int Laguerre_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,double sens,
                        int MaxIter,INTERVAL &Lower,INTERVAL &Upper);
```

In that case `Upper`=[a,b] will be such that value of all roots of any polynomial within the set is lower than b, while for some polynomial they will be lower than a. On the other hand `Lower`=[a,b] will be such that the value of all roots of any polynomial within the set is greater than a, while for some polynomial they will be greater than b.

5.2.5.3 Example

The procedure:

```
Coeff(1)= -3;Coeff(2)=2;Coeff(3)=-1;Coeff(4)=1;
Num=Laguerre_Bound_Interval(3,Coeff,0.1,50,Bound);
Coeff_App(1)= INTERVAL(-3.1,-2.9);Coeff_App(2)=INTERVAL(1.9,2.1);
Coeff_App(3)=INTERVAL(-1.1,-0.9);Coeff_App(4)=INTERVAL(0.9,1.1);
Num=Laguerre_Bound_Interval(3,Coeff_App,0.1,50,Lower,Upper);
```

leads to `Bound`=[1.15385,1.3], `Lower`=[1.08209,1.40271], `Upper`=[1.21818,1.52222].

5.2.6 Laguerre second method

5.2.6.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = x^n + a_1x^{n-1} + ...a_n = 0$$

Assume that P has n real roots. Then [23] all roots are contained in the interval whose end-point are given by the two solutions of the quadratic equation:

$$nx^2 + 2a_1x + [2(n-1)a_2 - (n-2)a_1^2] = 0$$

5.2.6.2 Implementation

This procedure enable to determine upper and lower bound on the roots of a polynomial if all the roots are real. In the implementation we check if all the roots of the polynomial are real using Huat theorem (see section 5.5.4) and if the answer is positive we determine the bounds. The syntax is:

```
int Laguerre_Second_Bound_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `Bound`: upper and lower bound on the roots

On success the return code is 1 while the return code is 0 if the Degree is lower than 2. A similar algorithm exists for interval polynomial:

```
int Laguerre_Second_Bound_Interval(int Degree,
      INTERVAL_VECTOR &Coeff,INTERVAL &Lower,INTERVAL &Upper);
```

In that case `Upper`=[a,b] will be such that the maximal value of all roots of any polynomial within the set is lower than b, while for some polynomial they will be lower than a. On the other hand `Lower`=[a,b] will be such that the minimal value of all roots of any polynomial within the set is greater than a, while for some polynomial they will be greater than b.

5.2.7 Newton method

5.2.7.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = a_0x^n + a_1x^{n-1} + ...a_n = 0$$

with $a_0a_n \neq 0$, $a_0 > 0$. Newton theorem state that if it exists c such that $P(c) > 0$ and for all the derivative $P^{(i)}$ of P, with $i \in [1, n]$, then c is an upper bound of the positive roots of P. To find c the following scheme can be used:

1. let c be such that $P^{(n-1)}(c) \geq 0$
2. let k the smallest integer such that either $k = n + 1$ or $k \leq n$ and $P^{(n-k)}(c) \leq 0$
3. if $k \leq n$ then substitute c by c' such that $P^{n-k)}(c') \geq 0$ and go to 2
4. return c

A consequence of Newton theorem is that the best bound cannot be lower than $-a_1/(a_0\ n)$.

5.2.7.2 Implementation

This procedure is able to give upper and lower bound on the value of the roots.

In the implementation we define c as the smallest real which satisfy $P^{(n-1)}(c) \geq 0$. If we find k such that $P^{(n-k)}(c) \leq 0$, then we increase c by a given positive value `sens` and start again. We limit the number of iteration of the scheme by giving a maximal value for the number of iteration:

```
int Newton_Bound_Interval(int Degree,VECTOR &Coeff1,double amp_sens,int MaxIter,double *bound);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree

- `sens`: a positive real indicating the increase of c in the scheme
- `MaxIter`: the maximum number of iteration. If this number is exceeded the procedure returns 0.
- `bound`: the upper value of the real root

This procedure fail and returns 0 if `Degree`=0, `Coeff(1)`=0, if `Coeff(Degree+1)`=0 and if the number of iteration exceed `MaxIter`. On success the return code is 1. Note that the bound given by Newton (assuming that `Coeff(Degree+1)` is positive) cannot be lower than `-Coeff(Degree)/(Degree Coeff(Degree+1))`. The lower bound of the root may be determined by:

```
int Newton_Bound_Inverse_Interval(int Degree,VECTOR &Coeff1,double amp_sens,
                      int MaxIter,double *bound);
```

We have also a procedure which determine upper and lower bound for the real roots:

```
int Newton_Bound_Interval(int Degree,VECTOR &Coeff1,double amp_sens,
                         int MaxIter,INTERVAL &Bound);
```

All the real roots lie within `Bound`. We may also use this procedure for interval polynomial:

```
int Newton_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,double sens,
                              int MaxIter,INTERVAL &Bound);
```

This procedure fail and returns 0 if `Degree`=0, $0 \in$ `Coeff(1)`, if $0 \in$`Coeff(Degree+1)` and if the number of iteration exceed `MaxIter`. If `Bound`=[a,b], then the real roots of all the polynomial in the set are lower than b and for some polynomial in the set the roots may be lower than a. A similar procedure exists for upper and lower bound.

```
int Newton_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,double sens,
                        int MaxIter,INTERVAL &Lower,INTERVAL &Upper);
```

In that case `Upper`=[a,b] will be such that value of all roots of any polynomial within the set is lower than b, while for some polynomial they will be lower than a. On the other hand `Lower`=[a,b] will be such that the value of all roots of any polynomial within the set is greater than a, while for some polynomial they will be greater than b.

5.2.8 Newton theorem

5.2.8.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = a_0x^n + a_1x^{n-1} + ...a_n = 0$$

with n real roots. The roots x of P are such that [4]:

$$x^2 < \left(\frac{a_1}{a_0}\right)^2 - 2\left(\frac{a_2}{a_0}\right)$$

5.2.8.2 Implementation

This procedure is able to give an upper bound on the absolute value of the roots as soon as all the roots are real (this is checked using Huat theorem, section 5.5.4). The syntax is:

```
int Newton_Second_Bound_Interval(int Degree,VECTOR &Coeff,double *bound);
```

with:

- `Degree`: degree of the polynomial

- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `bound`: upper bound on the absolute value of the roots

The procedure returns 0 if `Degree`=0 or `Coeff(Degree+1)`=0, 1 otherwise. We may also compute a lower bound on the absolute value of the root by using:

```
int Newton_Second_Bound_Inverse_Interval(int Degree,VECTOR &Coeff,double *bound);
```

An upper and lower bound on the absolute value of the roots may be compute by:

```
int Newton_Second_Bound_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
```

In that case if `Bound`=[a,b] we have for all root x: `a`$\leq |x| \leq$ `b`.

Similar procedures exist for interval polynomial:

```
int Newton_Second_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Bound);
int Newton_Second_Bound_Inverse_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Bound);
int Newton_Second_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &L, INTERVAL &U);
```

where

- `Bound`: upper or lower bound on the absolute value of the root of the polynomials
- `L`,`U`: if `L`=[a,b] then the absolute value of the roots of any polynomial in the set is greater than a while some polynomial have root greater than b. Conversely if `U`=[a,b] the absolute value of the roots of any polynomial in the set is lower than b while some polynomial have root lower than a.

5.2.9 Joyal bounds

5.2.9.1 Mathematical background

Let a polynomial whose interval coefficients have a fixed sign. Let a_n be the leading coefficient such that $a_n = 1$. We define the sequence $A_1 = |a_{n-1}a_0|$, $A_2 = |a_{n-1}a_1 - a0|$, $A_j = |a_{n-1}a_j - a_{j-1}|$. If A is the largest upper bound of the sequence A_j, then the modulus ρ of the roots of the polynomial is lower or equal to $(1+\sqrt{1+4A})/2$.

5.2.9.2 Implementation

```
int Joyal_Bound_Interval(int Degre,INTERVAL_VECTOR &Coeff,INTERVAL
&Bound)
```

returns in `Bound` an upper bound of the modulus of the roots of the polynomial of degree `Degre` and interval coefficients `Coeff`. The procedure returns 0 if a_n is not equal to 1, 1 otherwise.

5.2.10 Pellet method

5.2.10.1 Mathematical background

Let $f = a_n x^n + \ldots + a_0$ be a polynomial and let $f_p = |a_n|x^n + \ldots + |a_{p+1}x^{p+1}| - |a_p|x^p - \ldots - |a_0|$ be the associated polynomial. If f_p has two positive roots $\beta > \alpha$, then f has exactly p roots in the disk $|x| < \alpha$ and no roots in the domain $\alpha < |x| < \beta$.

Now assume that we have calculated the coefficients of the polynomial $f(x+a+R) = Q(X)$ where a, R are known quantities. If we determine a Q_p polynomial that has 2 positive roots a_1, a_2 in the range $[0, R]$, then $Q(x)$ has roots in the disk $|X| < a_1 < R$. Hence the absolute value of the real part of the roots is bounded by a_1. As $x = X - a - R$ we get that the real part R_p of the root satisfies $-a_1 + a + R \leq R_p < a_1 + a + R$. As $R > a_1$ this shows that f has roots whose real part is greater than a.

5.2.10.2 Implementation

```
int Pellet(int Degre,INTERVAL_VECTOR &Coeff,double R)
```

where `Degre` is the degree of the polynomial, `Coeff` are the coefficients of the polynomial $f(x + a + R)$. This procedure returns 1 if roots whose real part is greater than a.

5.2.11 Global implementation

The above algorithms have been regrouped in two procedures, one for determining the bound on the positive roots, the other for the negative roots:

```
int Global_Positive_Bound_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
int Global_Negative_Bound_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `Bound`: the bound on the positive or negative roots

For the positive root if `Bound`=[a,b] and b=0, then there is no positive root. Similarly for the negative root is a=0, then there is no negative root. The procedure returns 0 in case of failure which correspond to the failure of all the previous algorithms, 1 in case of success. Note that for Laguerre and Newton method the maximum number of iteration is defined by the global variable `Max_Iter_Laguerre_Interval` fixed by default to 1000. The step size is defined as:

- if a bound B have been previously determined, then the step size is fixed to $B/30$ except if the global variable `Step_Laguerre_Interval` has been defined to be a double not equal to 0, in which case this variable is used as the step size.
- if a bound as not been determined the step size is fixed to 1 except if the global variable `Step_Laguerre_Interval` has been defined to be a double not equal to 0, in which case this variable is used as the step size.
- if the step if lower than the global variable `Min_Step_Laguerre_Interval` (which is 0.1 by default) the step is substituted by `Min_Step_Laguerre_Interval`

Similar procedure exist for interval polynomial:

```
int Global_Positive_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,
                                   INTERVAL &Lower,INTERVAL &Upper);
int Global_Negative_Bound_Interval(int Degree,INTERVAL_VECTOR &Coeff,
                                   INTERVAL &Lower,INTERVAL &Upper);
```

where `Lower` is an interval on the lower bound: for positive roots and if `Lower`=[a,b] then the roots of any polynomial in the set is greater than a while some polynomial have root greater than b. Conversely if `Upper`=[a,b] the roots of any polynomial in the set is lower than b while some polynomial have root lower than a.

Both procedures for real roots have been regrouped in

```
void ALIAS_Find_Bound_Polynom(int Degree,
           INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
           INTERVAL_VECTOR &PALL,INTERVAL &Space, INTERVAL &Bound)
```

where

- `PALL`: the first component is the bound for the real roots the polynomial while the following are the value of the parameters for which the bounds will be obtained
- `Space`: a default bound for the real roots; may be used to indicate that we are looking only for bounds on the positive or negative real roots by specifying a positive or a negative lower bound
- `Bound`: bounds for the real roots

5.2.11.1 Example

An interesting example is given by:

```
Coeff_App(1)= INTERVAL(-3.1,-2.9);Coeff_App(2)=INTERVAL(1.9,2.1);
Coeff_App(3)=INTERVAL(-1.1,-0.9);Coeff_App(4)=INTERVAL(0.9,1.1);
Num=Global_Positive_Bound_Interval(3,Coeff_App,Lower,Upper);
```

which leads to `Lower`=[1.1487,1.42957], `Upper`=[1.15224,1.43049].

The above procedure may give some sharp bounds. For example consider the Wilkinson polynomial of order 22 (which has as roots 1,2,...,21,22) we get 0 negative roots while the positive roots are bounded by [0.790447,22.1087].

The test program `Test_Bound_UP` enable to test the bound procedures for any polynomial. This program take as first argument the name of a file giving the coefficients of the polynomial by increasing power of the variable. The program will print the bounds determined by all the previous procedures and then the bounds determined using the global implementation. Then the same treatment will be applied on the interval polynomial whose coefficients are intervals centered at the coefficients find in the file with a width of 0.2

5.2.12 Kantorovitch theorem

The mathematical background of Kantorovitch has been explained in section 3.1.2. This method may determine an interval on the unknown in which there is an unique solution, toward which Newton method will converge (see section 2.9).

5.2.12.1 Implementation

The syntax of the algorithm is:

```
int Kantorovitch(int Degree,VECTOR &Coeff,REAL Input,double *eps)
```

with

- `Degree`: degree of the polynomial
- `Coeff`: coefficients of the polynomial ordered along increasing degree
- `Input`: center of the interval in which an unique solution occur
- `eps`: half width of the solution interval

If this procedure returns 1, then there is an unique solution in the interval [`Input-eps,Input+eps`]. There is also an implementation of Kantorovitch theorem for interval polynomial:

```
int Kantorovitch(int Degree,INTERVAL_VECTOR &Coeff,REAL Input,double *eps)
```

If this procedure returns 1, then any polynomial in the set of interval polynomial has an unique solution in the interval [`Input-eps,Input+eps`]. There is also an implementation which take into account rounding errors:

```
int Kantorovitch_Fast_Safe(int Degree,INTERVAL_VECTOR &Coeff,REAL Input,double *eps)
```

in which "safe" interval value of the coefficients have been pre-computed.

5.2.12.2 Example

Let $P = x^3 - x^2 + 2x - 3$, `Input` be 1.2 and the procedure:

```
Coeff(1)= -3;Coeff(2)=2;Coeff(3)=-1;Coeff(4)=1;
Num=Kantorovitch(3,Coeff,1.2,&eps);
Coeff_App(1)= INTERVAL(-3.1,-2.9);Coeff_App(2)=INTERVAL(1.9,2.1);
Coeff_App(3)=INTERVAL(-1.1,-0.9);Coeff_App(4)=INTERVAL(0.9,1.1);
Num=Kantorovitch(3,Coeff_App,1.2,&eps);
```

In the first case `Kantorovitch` returns 1 and has determined that there is an unique solution in the interval [1.04082,1.35918]. Using Newton method we find the root 1.27568220371567 with residue 2.80147904874184e-10. For the interval polynomial `Kantorovitch` returns also 1 and find an unique solution in [1.05718,1.34282].

5.3 Bounds on the product and sum of roots

5.3.1 Newton relations

5.3.1.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = a_0x^n + a_1x^{n-1} + ...a_n = 0$$

and $x_1, \ldots, x_n$ the real and complex roots of P. Let define S_p as

$$S_p = \sum_{i=1}^{i=n} x_i^p$$

We have [4]:

$$\begin{aligned}
&a_0S_1 + a_1 = 0\\
&a_0S_2 + a_1S_1 + 2a_2 = 0\\
&\ldots\\
&a_0S_p + a_1S_{p-1} + a_2S_{p-2} + \ldots + a_{p-1}S_1 + pa_p = 0\\
&\ldots\\
&a_0S_n + a_1S_{n-1} + \ldots + a_{n-1}S_1 + a_n = 0
\end{aligned}$$

5.3.2 Implementation

Let x_i be the n roots (either complex or real) of a polynomial of degree n. Let S_p be:

$$S_p = \sum_{i=1}^{i=n} x_i^p$$

This procedure enable to compute the n elements $S_1, \ldots, S_n$. The syntax is:

```
VECTOR SumN_Polynomial_Interval(int Degree,VECTOR &Coeff)
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: coefficients of the polynomial ordered along increasing degree

This procedure returns 0 if the leading coefficient is equal to 0, 1 otherwise. There is an equivalent procedure for interval polynomial:

```
INTERVAL_VECTOR SumN_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff)
```

which returns intervals including the S_p. This procedure returns 0 if 0 is included in the leading interval.

5.3.3 Viète relations

5.3.3.1 Mathematical background

Let $P(x)$ be an univariate polynomial of degree n:

$$P(x) = a_0x^n + a_1x^{n-1} + ...a_n = 0$$

and $x_1, \ldots, x_n$ the real and complex roots of P. Let define Z_p as:

$$Z_p = \sum_{i<j<\ldots<k, i=1,\ldots,n} x_ix_j\ldots x_k$$

We have [4]:

$$\begin{aligned}
&a_0 Z_1 + a_1 = 0\\
&a_0 Z_2 - a_2 = 0\\
&\ldots\\
&a_0 Z_k + (-1)^k a_k = 0\\
&\ldots\\
&a_0 \prod_{i=1}^{i=n} x_i + (-1)^n a_n = 0
\end{aligned}$$

5.3.4 Implementation

Let x_i be the n roots (either complex or real) of a polynomial of degree n. Let Z_p be:

$$Z_p = \sum_{i<j<\ldots<k, i=1,\ldots,n} x_i x_j \ldots x_k$$

This procedure enable to compute the n elements $Z_1, \ldots, Z_n$ (Z_1 is the sum of the roots, Z_n the product of the roots). The syntax is:

```
VECTOR ProdN_Polynomial_Interval(int Degree,VECTOR &Coeff)
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: coefficients of the polynomial ordered along increasing degree

This procedure returns 0 if the leading coefficient is equal to 0, 1 otherwise. There is an equivalent procedure for interval polynomial:

```
INTERVAL_VECTOR ProdN_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff)
```

which returns intervals including the Z_p. This procedure returns 0 if 0 is included in the leading interval.

5.4 Maximum number of real roots

5.5 Number of real roots

5.5.1 Descartes Lemma

5.5.1.1 Mathematical background

Let P the polynomial:

$$P = a_0 + a_1 x + \ldots + a_n x^n$$

with $a_0 a_n \neq 0$. Let the sequence $\{a_0, a_1, \ldots, a_n\}$ and the n the number of change of sign in this sequence. Then the number of positive real roots, counted with their order of multiplicity is equal to $n-2k$ with $k \in [0, n/2]$ [13].

5.5.1.2 Implementation

The syntax of the procedure is:

```
INT Descartes_Lemma_Interval(int Degree,VECTOR &Coeff)
```

with:

- `Degree`: degree of the polynomial

- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree

This procedure returns the number of positive real roots up to an even number. If the procedure returns m the number of positive roots is $m-2k$ with $k \in [0, m/2]$.

There is an implementation of this method for interval polynomial. Here it is necessary to introduce an additional parameter to indicate the confidence we have in the result. The procedure is implemented as:

```
INT Descartes_Lemma_Interval(int Degree,INTERVAL_VECTOR &Coeff,int *Confidence);
```

`Confidence` is a quality index:

- 1: the result is exact, for all polynomials in the set the number of positive real roots is $m-2k$
- ≤ 0: the number of positive real roots for all the polynomials in the set is $\leq m$

Similar algorithms for negative roots are available:

```
INT Descartes_Lemma_Negative_Interval(int Degree,VECTOR &Coeff)
INT Descartes_Lemma_Negative_Interval(int Degree,INTERVAL_VECTOR &Coeff,int *Confidence)
```

5.5.2 Budan-Fourier method

5.5.2.1 Mathematical background

Budan-Fourier algorithm is a simple method which enable to determine easily some information on the number of root of a given univariate polynomial within a given interval. Let P the polynomial:

$$P = a_0 + a_1 x + \ldots + a_n x^n$$

and $P^{(n)}$ its n-th derivative. Let the interval $(\underline{x}, \overline{x})$ the interval in which we are looking for roots. We assume that $P(\underline{x}) \neq 0$, $P(\overline{x}) \neq 0$ and $a_0 \neq 0$. We construct the sequence $L = \{P(\underline{x}), P^{(1)}(\underline{x}), \ldots, P^{(n)}(\underline{x})\}$ from which we exclude the 0 element. Similarly we construct the sequence $U = \{P(\overline{x}), P^{(1)}(\overline{x}), \ldots, P^{(n)}(\overline{x})\}$ (a special treatment has to be applied for the zero element of U, see [13]). Let $\underline{N}$ the number of change of sign in L and $\overline{N}$ the number of change of sign in U. Then the number of real roots of P in $(\underline{x}, \overline{x})$, counted with their order of multiplicity, is $\underline{N} - \overline{N}$ or lower than this number by an even number.

5.5.2.2 Implementation

This procedure is used to determine the number of real roots in a given interval, up to an even number. The syntax of the procedure is:

```
INT Budan_Fourier_Interval(int Degree,VECTOR &Coeff,INTERVAL In)
INT Budan_Fourier_Interval(int Degree,INTEGER_VECTOR &Coeff,INTERVAL In)
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree, which may be `REAL` or `INT`
- `In`: the interval in which we are looking for the number of roots

If this procedure returns the integer $m \geq 0$, then the number of real roots in `In` is $m-2k$ with $k \in [0, m/2]$. A negative returns code indicate a failure of the algorithm:

- -1: $P(\underline{\texttt{In}}) = 0$
- -2: $a_0 = 0$, the polynomial may be factored
- -3: $P(\overline{\texttt{In}}) = 0$

This procedure may be used with polynomial whose coefficients are intervals. The syntax is:

```
INT Budan_Fourier_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL In,int *Confidence)
```

where `Confidence` is a quality index for the result:

- 1: the result is exact, so if m is the return code of the algorithm the number of reals roots of the interval polynomial is $m-2k$ with $k \in [0, m/2]$
- ≤ 0: if m is the return code of the algorithm the number of reals roots of the interval polynomial is $\leq m$

Due to rounding errors incorrect results may be returned by the previous procedures. A safer procedure is:

```
INT Budan_Fourier_Safe_Interval(int Degree,VECTOR &Coeff,
                                INTERVAL In,INTERVAL &NbRoot);
```

The procedure returns 1 in case of success and an interval for the number of roots. If `NbRoot`=[a,b], then if a =b the number of roots is either a,a-2,... and if a $\neq$ b the number of roots is lower than b. If "safe" value of the coefficients have been pre-computed you may use:

```
INT Budan_Fourier_Fast_Safe_Interval(int Degree,INTERVAL_VECTOR &Coeff,
                                     INTERVAL In,INTERVAL &NbRoot);
```

Another safe procedure is:

```
INT Budan_Fourier_Interval(int Degree,INTEGER_VECTOR &Coeff,int Inf,int Sup)
```

where the coefficients and the bounds are integers.

5.5.2.3 Example

Let $P = x^3 - x^2 + 2x - 3$, `In` be the interval [0,2] and the procedure:

```
Coeff(1)= -3;Coeff(2)=2;Coeff(3)=-1;Coeff(4)=1;
P=INTERVAL(0.0,2.0);
Num= Budan_Fourier_Interval(3,Coeff,P);
```

`Num` is 3 meaning that P has either 3 or 1 roots in the interval. Now assume that we have an interval polynomial:

```
Coeff_App(1)= INTERVAL(-3.1,-2.9);
Coeff_App(2)=INTERVAL(1.9,2.1);
Coeff_App(3)=INTERVAL(-1.1,-0.9);
Coeff_App(4)=INTERVAL(0.9,1.1);
Num= Budan_Fourier_Interval(Degree,Coeff_App,P,&Confidence);
```

`Num` is also 3 with `Confidence`=1 meaning that the number of roots for any polynomial is either 3 or 1.

5.5.3 Sturm method

5.5.3.1 Mathematical background

Let a polynomial P and define $f_0 = P$, f_1 be the first derivative of P. Then define the sequence $f_i = -\texttt{Rem}(f_{i-2}, f_{i-1})$ where `Rem` is the remainder of the division of f_{i-2} by f_{i-1}. If n is the degree of P the last element in the sequence will be f_n. Assume now that we are looking for the number of distinct roots of P in the interval $[a, b]$ with $P(a)P(b) \neq 0$. We build two sequences:

$$\{f_0(a), f_1(a), \ldots, f_n(a)\} \quad \{f_0(b), f_1(b), \ldots, f_n(b)\}$$

Let n_0 be the number of change of sign in the first sequence and n_1 be the number of change of sign in the second sequence. Then the number of distinct real roots of P in the interval $[a, b]$ is $n_0 - n_1$ [14] if $f_0(a)f_0(b) \neq 0$. Note that a multiple roots count only for one root with this method.

The drawback of Sturm method is that the absolute value of the coefficients increase quickly when computing the sequence. Numerical rounding errors may then affect the result. The alternate method of Budan-Fourier (see section 5.5.2) is less sensitive to rounding errors although it provides less information than Sturm method.

5.5.3.2 Implementation

This procedure determines the number of real roots of an univariate polynomial in a given interval. It is implemented as:

```
int Sturm_Interval(int Degree,VECTOR &Coeff,INTERVAL &In);
int Sturm_Interval(int Degree,INTEGER_VECTOR &Coeff,INTERVAL &In);
```

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree, either real or integers
- `In`: the interval in which we are looking for the number of roots

This procedure returns 0 or a positive number on success, this number being the number of roots of the polynomial in the interval. A return code of -1 means that either the polynomial is equal to 0, is equal to 0 at one of the extremity of the interval or has multiple roots. However Sturm method is sensitive to numerical rounding errors. For high degree polynomial it may be better to use the "safer" procedure:

```
int Sturm_Safe_Interval(int Degree,VECTOR &Coeff,INTERVAL &In,INTERVAL &NbRoot);
```

which returns 1 in case of success and an interval `NbRoot` which contain the number of roots. If `NbRoot`=[a,b], then if a =b the number of roots is a and if a $\neq$ b the number of roots is lower than b. Another safe procedure can be used if the coefficients are integers and the interval is also defined by integer numbers:

```
int Sturm_Interval(int Degree,INTEGER_VECTOR &Coeff,int Inf,int Sup);
```

This procedure will return `-1` if all the coefficients are 0 and `-2` if at some point of the process an integer larger than the largest machine integer is encountered (**Not yet implemented**).

5.5.3.3 Example

We use as example the Wilkinson polynomial of degree n where P:

$$P = \prod_{i=1}^{i=n}(x-i)$$

and we are looking for roots in the interval [0.5,0.7] the procedure `Sturm_Interval` returns always 0 for order 4 to 20. But starting at order 9 the safe procedure returns [0,3] while the safe Budan-Fourier procedure still returns 0.

The test program `Test_Nb_Root_Up` enable to test Budan-Fourier and Sturm method on any polynomial whose coefficients are defined in a file, by increasing degree of the power of the unknown.

5.5.4 Du Gua-Huat-Euler theorem

5.5.4.1 Mathematical background

Let P the polynomial:

$$P = a_0 + a_1x + \ldots + a_nx^n$$

If all the roots of P are real then [13]:

$$a_i^2 \geq a_{i-1}a_{i+1} \quad \text{for } i \in [1,n]$$

Conversely if this relation does not hold for some i (for example if a coefficient is 0 while its neighbor have same sign), then there are complex roots.

5.5.4.2 Implementation

This procedure enable to determine if the roots of a polynomial are all real. It is implemented in a safe way i.e. if they are numerical errors in the coefficients and if the procedure determine that all the roots are real the result is guaranteed. The syntax is:

```
int Huat_Polynomial_Interval(int Degree,VECTOR &Coeff);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree

This procedure returns 1 if all the roots are real, 0 otherwise. There is a similar procedure for interval polynomial:

```
int Huat_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff);
```

This procedure will return 1 if all the polynomials in the set have their roots real, `-1` if some polynomials in the set have all roots real but some others have complex roots and 0 if all the polynomials in the set have complex roots.

5.6 Separation between the roots

5.6.1 Rump theorem

5.6.1.1 Mathematical background

Let P the polynomial:

$$P = a_0 + a_1x + \ldots + a_nx^n$$

and let $|P|$ be:

$$|P| = \mathrm{Max}_{i=0}^{i=n}|a_i|$$

The minimal distance Δ between the real roots of P is such that [26]:

$$\Delta > \sqrt{\frac{8}{n^{n+2}}}\frac{1}{1+|P|^n}$$

5.6.1.2 Implementation

The procedure is able to determine a lower bound on the distance between two real roots of a polynomial:

```
int Min_Sep_Root_Interval(int Degree,VECTOR &Coeff,double &min);
```

with

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients of the polynomial in increasing degree
- `min`: the lower bound on the distance between two real roots

Similarly an upper bound may be determined with:

```
int Max_Sep_Root_Interval(int Degree,VECTOR &Coeff,double &max);
```

while upper and lower bounds may be determined with:

```
int Bound_Sep_Root_Interval(int Degree,VECTOR &Coeff,INTERVAL &Bound);
```

There is also a procedure to determine a lower bound for interval polynomial:

```
int Min_Sep_Root_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL &Lower);
```

If `Lower`=[a,b] then some polynomials in the set will have a minimal distance between the roots greater than b while all the polynomials in the set have a minimal distance greater than a.

5.6.1.3 Example

Clearly this formula give an underestimated value of the minimal distance between the roots. For example if we consider a Wilkinson polynomial of order 4 (which has therefore 1,2,3,4 as roots) we find that the minimal and maximal distance between the roots are bounded by [7.07107e-09,1.41421e+08].

5.7 Analyzing the real roots

The procedure

```
int ALIAS_Min_Max_Is_Root(int Degree,
  int NbParameter,
  int HasInterval,
  INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
  int Iteration,
  INTERVAL_VECTOR &Par,
  double *Root,
  int Type,
  int (* Solve_Poly)(double *, int *,double
*))
```

may be used to rest if a polynomial in a set may have as real root one of two pre-defined value.

- `Degree`: the degree of the polynomial
- `NbParameter`: the number of parameters that appear in the coefficient of the polynomial
- `HasInterval`: 1 if the coefficient include intervals not defined by parameters, 0 otherwise
- `TheCoeff`: a procedure to calculate the coefficients of the polynomial being given range for the parameters
- `Iteration`: the maximum number of box that may be used by the algorithm
- `Par`: the ranges for the parameters
- `Root`: we look for polynomial whose real part of the root is either Root[0] or Root[1]
- `Type`:
  - -1 : if in a box we have found a polynomial with a root lower than Root[0]; we look for a polynomial in the box whose real root is exactly Root[0]. If no such polynomial is found the box is eliminated
  - 1 : if in a box we have found a polynomial with a root greater than Root[1]; we look for a polynomial in the box whose real root is exactly Root[1]. If no such polynomial is found the box
  - 2: if in a box we have found a polynomial with a root greater than Root[1] and a polynomial with a real root greater than Root[0]; we look for a polynomial in the box whose real root is exactly Root[1] and a polynomial whose real root is exactly Root[0]. If no such polynomials are found the box is eliminated
  - 0: general case;
- `Solve_Poly`: a procedure to solve polynomial with double floating point coefficients. The first argument are the coefficients, the second the degree of the polynomial and the third the real roots

This procedure returns -1 if no polynomial with real root `Root` exist, 1 if such polynomial exist and 0 if the algorithm has not be able to determine such polynomial

5.8 Analyzing the real part of the roots

For some applications it may be interesting to determine if a polynomial in a set defined by a parametric polynomial has the real part of one of its roots equal to a pre-defined value. This may be done by using the procedure

```
int ALIAS_Is_Root_RealPart(int Degree,
           int NbParameter,
           int HasInterval,
           INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
           INTERVAL_VECTOR (* TheCoeffCentered)(INTERVAL_VECTOR&,double a),
           int Iteration,
           INTERVAL_VECTOR &Par,
           double *Root,
           INTERVAL_VECTOR (* EvaluateComplex)(int,int,INTERVAL_VECTOR &),
           int (* Simp)(INTERVAL_VECTOR &))
```

where

- `Degree`: the degree of the polynomial
- `NbParameter`: the number of parameters that appear in the coefficient of the polynomial
- `HasInterval`: 1 if the coefficient include intervals not defined by parameters, 0 otherwise
- `TheCoeff`: a procedure to calculate the coefficients of the polynomial being given range for the parameters
- `TheCoeffCentered`: a procedure to calculate the coefficients of the polynomial $P(x + a)$
- `Iteration`: the maximum number of box that may be used by the algorithm
- `Par`: the ranges for the parameters
- `Root`: we look for polynomial whose real part of the root is either Root[0] or Root[1]
- `EvaluateComplex`: a procedure that returns 4 intervals. The input interval vector has dimension `NbParameter+1` elements, the first one being the parameters, the last one being b. The procedure should evaluate $U = P(Root[0] + Ib), V = P(Root[1] + Ib)$ and should return the real part of U, the complex part of U, the real part of V and the complex part of V
- `Simp`: an optional simplification procedure that returns -1 if the polynomials in a set cannot roots with real part equal to Root[0] or Root[1]

5.9 Utilities

5.9.1 Addition of two polynomials

The following procedures enable to add polynomials with real,interval or integer coefficients:

```
VECTOR Add_Polynomial_Interval(int n1,VECTOR &Coeff1,int n2,VECTOR &Coeff2);
INTERVAL_VECTOR Add_Polynomial_Interval(int n1,INTERVAL_VECTOR &Coeff1,
                  int n2,INTERVAL_VECTOR &Coeff2);
INTEGER_VECTOR Add_Polynomial_Interval(int n1,INTEGER_VECTOR &Coeff1,
      int n2,INTEGER_VECTOR &Coeff2);
```

with:

- `ni`: degree of the polynomial
- `Coeffi`: the coefficients of the polynomials in increasing degree

There is also a version which take into account rounding errors:

```
INTERVAL_VECTOR Add_Polynomial_Safe_Interval(int n1,VECTOR &Coeff1,int n2,VECTOR &Coeff2);
```

5.9.2 Multiplication of two polynomials

The following procedures returns the degree of the product of two polynomial either with real or integer coefficients:

```
int Degree_Product_Polynomial_Interval(int n1,VECTOR &Coeff1,int n2,VECTOR &Coeff2);
int Degree_Product_Polynomial_Interval(int n1,INTEGER_VECTOR &Coeff1,int n2,
      INTEGER_VECTOR &Coeff2);
```

Then you may use the following procedure to compute the product of polynomial either with real,integer or interval coefficients:

```
VECTOR Multiply_Polynomial_Interval(int n1,VECTOR &Coeff1,int n2,VECTOR &Coeff2);
INTEGER_VECTOR Multiply_Polynomial_Interval(int n1,INTEGER_VECTOR &Coeff1,
      int n2,INTEGER_VECTOR &Coeff2);
INTERVAL_VECTOR Multiply_Polynomial_Interval(int n1,INTERVAL_VECTOR &Coeff1,
      int n2,INTERVAL_VECTOR &Coeff2);
```

All these procedures return the coefficients of the product, the leading term being 0 only if one polynomial is equal to 0.

To take into account the rounding errors you may use:

```
INTERVAL_VECTOR Multiply_Polynomial_Safe_Interval(int n1,VECTOR &Coeff1,int n2,VECTOR &Coeff2);
```

5.9.3 Evaluation of a polynomial

The evaluation of a polynomial for a given value is implemented as:

```
REAL Evaluate_Polynomial_Interval(int Degree,VECTOR &Coeff,REAL P)
INTERVAL Evaluate_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff,REAL P)
INTERVAL Evaluate_Polynomial_Interval(int Degree,VECTOR &Coeff,INTERVAL P)
INTERVAL Evaluate_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL P)
REAL Evaluate_Polynomial_Interval(int Degree,INTEGER_VECTOR &Coeff,REAL P);
int Evaluate_Polynomial_Interval(int Degree,INTEGER_VECTOR &Coeff,INT P);
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients (which can be `REAL`, `INT` or `INTERVAL` of the polynomial in increasing degree
- `P`: the point at which we want to compute the polynomial. It may be `REAL`, `INT` or `INTERVAL`.

These procedures enable to get:

- the value of a polynomial with `REAL` coefficients at a point
- the interval value of a polynomial with interval coefficients at a real point
- the interval value of a polynomial with `REAL` coefficients at for an interval value of the unknown. The first and second order derivative of the polynomial are used to get sharp bounds
- the interval value of a polynomial with interval coefficients for an interval value of the unknown. The first and second order derivative of the polynomial are used to get sharp bounds

5.9.3.1 Evaluation in centered form

The previous procedures may yield sometime to a bad evaluation of the polynomial. For example for the Wilkinson polynomial at order 15 the evaluation for 15.1 leads to 11977410100.791748046875, the correct value being 11977396665.006561033796551. The evaluation obtained when considering that the coefficients and the unknown are intervals is evidently correct and leads to the interval:

```
[11977379510.705810546875,11977410100.791748046875]
```

A better evaluation may be obtained if we use a centered form of the polynomial. Consider the polynomials:

$$P = a_0 + a_1x + \ldots + a_nx^n$$
$$P_1 = b_0 + b_1(x - c) + \ldots + b_n(x - c)^n$$

where c is a real. For an appropriate choice of the b_i we may have $P = P_1$ for all x. The procedures:

```
VECTOR Coeff_Polynomial_Centered_Interval(int Degree,VECTOR &Coeff,REAL P);
INTERVAL_VECTOR Coeff_Polynomial_Centered_Interval(int Degree,
                  INTERVAL_VECTOR &Coeff,REAL P);
```

enable to compute the b_i for the centered form at `P` either for polynomial or interval polynomial. To take into account numerical errors you may use:

```
INTERVAL_VECTOR Coeff_Polynomial_Centered_Safe_Interval(int Degree,VECTOR &Coeff,REAL P);
INTERVAL_VECTOR Coeff_Polynomial_Centered_Fast_Safe_Interval(int Degree,
           INTERVAL_VECTOR &Coeff,REAL P);
```

which return safe value for the coefficients (in the second form we assume that you have pre-computed safe value for the coefficients of the polynomial using the procedure described in section 5.9.10).

Then we may use the procedures:

```
REAL Evaluate_Polynomial_Centered_Interval(int Degree,VECTOR &Coeff,REAL Center,REAL P);
INTERVAL Evaluate_Polynomial_Centered_Interval(int Degree,VECTOR &Coeff,INTERVAL P);
INTERVAL Evaluate_Polynomial_Centered_Interval(int Degree,INTERVAL_VECTOR &Coeff,
    REAL Center,REAL P);
INTERVAL Evaluate_Polynomial_Centered_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL P);
```

These procedures return the evaluation of the polynomial at `P` using the centered form at `Center` or at the middle point of `P` if is an interval. For example for the Wilkinson polynomial at order 15 the evaluation for 15.1 using the centered form at 15 leads to 11977396665.00650787353516 which is largely better than the previous evaluation.

5.9.3.2 Safe evaluation of a polynomial

Due to the numerical error in the coefficients and in x the evaluation of a polynomial at x may be incorrect. The following procedures enable to compute an interval which is guaranteed to include the true value:

```
INTERVAL Evaluate_Polynomial_Safe_Interval(int Degree,VECTOR &Coeff,REAL P);
INTERVAL Evaluate_Polynomial_Safe_Interval(int Degree,VECTOR &Coeff,INTERVAL P);
INTERVAL Evaluate_Polynomial_Safe_Interval(int Degree,INTERVAL_VECTOR &Coeff,REAL P);
INTERVAL Evaluate_Polynomial_Safe_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL P);
```

For example for the Wilkinson polynomial at order 15 the evaluation for 15.1 leads to the interval:

$$[11977320583.22778129577637, 11977474081.67480659484863].$$

If "safe" intervals have been pre-computed (for example by using the procedure described in section 5.9.10) for the coefficients you may use:

```
INTERVAL Evaluate_Polynomial_Fast_Safe_Interval(int Degree,INTERVAL_VECTOR &Coeff,REAL P);
INTERVAL Evaluate_Polynomial_Fast_Safe_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL P);
```

which are faster. If you want to use a centered form you may use:

```
INTERVAL Evaluate_Polynomial_Centered_Safe_Interval(int Degree,VECTOR &Coeff,REAL P);
INTERVAL Evaluate_Polynomial_Centered_Safe_Interval(int Degree,VECTOR &Coeff,INTERVAL P);
INTERVAL Evaluate_Polynomial_Centered_Fast_Safe_Interval(int Degree,
        INTERVAL_VECTOR &Coeff,REAL P);
INTERVAL Evaluate_Polynomial_Centered_Fast_Safe_Interval(int Degree,
       INTERVAL_VECTOR &Coeff,INTERVAL P);
```

in which the two last forms assume that you have pre-computed a safe value of the coefficients of the polynomial.

The evaluation of a polynomial with real coefficients at a complex point may be performed with:

```
INTERVAL_VECTOR Evaluate_Complex_Poly(int deg,
        INTERVAL_VECTOR &Coeff, INTERVAL &XR,INTERVAL &XI)
```

where `XR` is the real part of the point and `XI` its imaginary part. This procedure returns an interval vector whose first element is the real part of the evaluation and the second element its imaginary part.

5.9.4 Sign of a polynomial

This procedure is able to determine if the sign of a polynomial evaluated at a given point may be affected by numerical errors.

5.9.4.1 Implementation

```
int Sign_Polynomial_Interval(int Degree,VECTOR &Coeff,REAL P)
int Sign_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff,REAL P)
int Sign_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff,REAL P)
int Sign_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff,INTERVAL P)
```

with:

- `Degree`: degree of the polynomial
- `Coeff`: the `Degree+1` coefficients (which can be `REAL` or `INTERVAL` of the polynomial in increasing degree
- `P`: the point or interval at which we want to compute the polynomial

This procedure returns 1 if the polynomial is positive, -1 if it is negative, 0 if it is 0 and 2 if the sign is either affected by numerical errors (if `Coeff` and `P` are `REAL`) or is not constant (if `Coeff` or `P` are `INTERVAL_VECTOR`).

5.9.5 Derivative of a polynomial

The derivatives of a polynomial may be computed using the procedures:

```
VECTOR Derivative_Polynomial_Interval(int Degree,VECTOR &Coeff)
INTEGER_VECTOR Derivative_Polynomial_Interval(int Degree,INTEGER_VECTOR &Coeff);
VECTOR Nth_Derivative_Polynomial_Interval(int Degree,VECTOR &Coeff,int n)
```

These procedures enable to get the coefficients of the first and `n`-th derivative of a polynomial defined by its `REAL` or `INT` coefficients.

Due to rounding errors there may be errors in the coefficients provided by the previous procedures. The procedure:

```
INTERVAL_VECTOR Derivative_Polynomial_Safe_Interval(int Degree,VECTOR &Coeff);
```

will return an `INTERVAL_VECTOR` (i.e. a set of intervals) which are guaranteed to include the true value of the coefficients of the derivative. A faster procedure may be used if "safe" interval values of the coefficients have been pre-computed:

```
INTERVAL_VECTOR Derivative_Polynomial_Fast_Safe_Interval(int Degree,INTERVAL_VECTOR &Coeff);
```

If we deal with polynomial with interval coefficients the following procedures may be used:

```
INTERVAL_VECTOR Derivative_Polynomial_Interval(int Degree,INTERVAL_VECTOR &Coeff)
INTERVAL_VECTOR Nth_Derivative_Polynomial_Interval(int Degree,
                                    INTERVAL_VECTOR &Coeff,int n)
```

5.9.6 Euclidian division

The procedure:

```
void ALIAS_Euclidian_Division(int Degree,INTERVAL_VECTOR &Coeff1,
     INTERVAL_VECTOR &Coeff2,INTERVAL &Residu,INTERVAL &a)
```

computes the division of the polynomial `P` with interval coefficients `Coeff1` by `x-a` and returns the polynomial `P1` with coefficients `Coeff2` such that `P=(x-a)P1+Residu`

Another procedure that perform the same process if `a` is a double is

```
void Divide_Single(int Degree,INTERVAL_VECTOR &Coeff1,double a,
                   INTERVAL_VECTOR &Coeff2, INTERVAL &Residu)
```

For dividing a polynomial P by the product of polynomials of type $(x-a_1)(x-a_2)\ldots(x-a_n)$ so that

$$P(x) = (x-a_1)(x-a_2)\ldots(x-a_n)Q(x) + R(x)$$

and evaluating the polynomial under this form for some interval X for x you may use

```
INTERVAL Divide_Evaluate_Multiple(int Degree,INTERVAL_VECTOR &Coeff1,
                                  int n,double *a,INTERVAL_VECTOR &X)
```

where

- `n`: the number of terms a_i
- `a`: the value of $a_1, \ldots, a_n$

The interval returned by this procedure is the the interval evaluation of P for x having the the interval value $X(1)$.

For the Euclidian division by an arbitrary polynomial you may use

```
int Divide_Polynom(int Degree,INTERVAL_VECTOR &Coeff,int DegreeDivisor,
                   INTERVAL_VECTOR &CoeffDivisor,
   INTERVAL_VECTOR &CoeffQuo, INTERVAL_VECTOR &CoeffRem)
```

The polynomial P of degree `Degree` with cofficients `Coeff` will be divided by the polynomial S of degree `DegreeDivisor` with coefficients `CoeffDivisor` so that the coefficient will be a polynomial Q of degree `Degree-DegreeDivisor` with coefficients `CoeffQuo` with a reminder which is a polynomial of degree `DegreeDivisor` and coefficients `CoeffRem`. This procedure returns 0 if the division is not possible (the interval coefficient of the leading term of S includes 0) or if the degree of S is greater than the degree of P. If the division has been performed this procedure will return 1.

Note that `Divide_Polynom` and `Divide_Evaluate_Multiple` may be used to perform the deflation of an univariate polynomial as soon as approximate roots for the polynomial have been found. This may be useful to decrease the computation time for solving a polynomial or for numerically instable polynomial such as the Wilkinson polynomial. A combination of Rouche filtering and of deflation allows to solve Wilkinson polynomial of order 18, while the general solving procedure will fail starting at order 13.

A special Euclidian division is the division of a polynomial P by its derivative. This can be done with

```
void Quotient_UP_Derivative(int Degree,VECTOR &Coeff,VECTOR &CoeffD,VECTOR &Quo,
                            VECTOR &Rem)
```

where P has coefficients `Coeff`, its derivative `CoeffD` and the coefficient of the division is `Quo` (the quotient is a polynomial of degree 1) while the coefficients of the remainder are stored in `Rem`

5.9.7 Expansion of $(x-a)^n$

The coefficients of the expansion of the polynomial $(x-a)^n$ may be obtained with the procedure

```
INTERVAL_VECTOR Power_Polynomial(INTERVAL &a,int n)
```

which returns the coefficients

5.9.8 Centered form

Let $P = sum_{i=0}^{i=n} a_i x^i$ which may also be written as $sum_{i=0}^{i=n} b_i (x-a)^i$ where a is some fixed or interval value. The coefficient b_i may be calculated with the procedure

```
int Derive_Polynomial_Expansion(int n,INTERVAL_VECTOR &Coeff,INTERVAL &a,INTERVAL_VECTOR &B)
```

which returns 1 if the b_i have been successfully calculated (0 otherwise). The coefficients of P are `Coeff` and the b_i are stored in `B`.

5.9.9 Unitary polynomial

Let P be a polynomial and be `maxroot` the maximal modulus of the root of P. From P we may derive a 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. The calculation of the coefficients of Q may be done with the procedure

```
int Unit_Polynom(int Degree,INTERVAL_VECTOR &Coeff,
                double maxroot,INTERVAL_VECTOR &CoeffU)
```

where `Coeff` are the coefficients of P and `CoeffU` the coefficients of Q. This procedure returns 1 if the calculation has been successfull, 0 otherwise.

5.9.10 Safe evaluation of a vector

Let us consider a `VECTOR`: due to numerical rounding each element of this vector represent in fact an interval. The procedure:

```
INTERVAL_VECTOR Evaluate_Coeff_Safe_Interval(int n,VECTOR &Coeff);
```

with:

- `n`: size of the vector
- `Coeff`: the vector

returns an `INTERVAL_VECTOR` guaranteed to include the true value of the elements of the vector.

Chapter 6

Parametric polynomials and eigenvalues of parametric matrices

A *parametric polynomial* is a polynomial whose coefficients are functions of a set of parameters (in other words it is a set of polynomials). A typical parametric polynomial is obtained when calculating the characteristic polynomial of a parametric matrix.

In this chapter we propose some algorithms to deal with the real roots of a parametric polynomial and, in some cases, with the real part of these roots. If the considered polynomial is the characteristic polynomial of a matrix we may make use of the components of the matrix.

Some of these algorithms use a primary and secondary algorithms. The secondary algorithm uses also a list of boxes which is stored in the interval matrix `BoxUP`.

6.1 Minimal and maximal real roots of a parametric polynomial

The purpose here is to determine the minimal and maximal values of the set of real roots of the the set of polynomials, when the parameters are bounded. There may be eventually also constraints on the parameters. The algorithm is implemented as:

```
int ALIAS_Min_Max_EigenValues(int Degree,
                              int Nb_Parameter,
                              INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
                              int Nb_Constraints,
                              INTEGER_VECTOR &Type_Eq,
                              int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
                              int Has_Matrix,
                              INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
                              int *Has_Gradient,
                              INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
                              INTERVAL & TheDomain,
                              INTERVAL_VECTOR & TheDomain_Parameter,
                              int Type,int Nb_Points,int Use_Solve,int rand,
                              int M,
                              double Accuracy_Variable,double Accuracy,double AccuracyM,
                              INTERVAL &Lowest_Root,INTERVAL &Highest_Root,
                              INTERVAL_MATRIX &Place,int  Stop,double *Seuil,
                              int (* Solve_Poly)(double *, int *,double *),
                              int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where the arguments are:

- `Degree`: degree of the polynomial

- `Nb_Parameter`: number of parameters appearing in the coefficients
- `TheCoeff`: a procedure that take as input an interval vector and returns the interval value of the coefficients of the polynomial. The elements of the input interval vector are first a range for the unknown in the polynomial, then the ranges for the `Nb_Parameter` parameters
- `Nb_Constraints`: the number of constraints expression that constraints the parameters
- `Type_Eq`: type of the constraints expressions (0 for equality, -1 for inequality of type ≤ 0, 1 for inequality of type ≥ 0)
- `TheMatrix`: the polynomial may be the characteristic polynomial of a matrix A. This argument is a procedure that takes as first argument the same interval vector as `TheCoeff` and returns in the second argument the interval component of A. This procedure must return 1 if the components have been successfully calculated, -1 otherwise
- `Has_Matrix`: this flag must be set to 0 except if the polynomial is the the characteristic polynomial of a matrix in which case it must be set to 1 (2 if the matrix is symmetrical)
- `IntervalFunction`: a function which return the interval vector evaluation of the constraints and of the polynomial, the last component of the vector being the interval evaluation of the polynomial. This procedure must be written in ALIAS standard form, see note 2.3.4.3
- `Has_Gradient`: an array of integer that indicates if the derivatives of the expression are available. Only the first element is used for now with the following value:
  - 0: no derivative available
  - 1: only the derivatives of the constraints are available, not the one of the polynomial
  - 2: all derivatives are available
- `Gradient`:a procedure which returns the Jacobian matrix of the expression for given values of the unknowns written in standard ALIAS form (see note 2.4.2.2)
- `TheDomain`: range for the polynomial unknown. This range must be large and is automatically adjusted during the calculation
- `TheDomain_Parameter`: the ranges for the parameters
- `Type`: 0 for finding only the minimum of the real root, 1 to find only the maximal root and 2 to find both
- `Nb_Points`: to estimate the minimal and maximal real root the algorithm compute the root of the polynomial at some given points for which the parameters have a fixed value. This value give the number of points where this procedure is used, it must be at least 1.
- `Use_Solve`: if this parameter is 1 or 3, then for each box we try to determine bounds for the real roots using algebraic geometry. If it is 2 or 3 then for each box we solve numerically the polynomial for some specific values of the parameters to update the minimum and maximum. If the value is 0 or 1 we will assume that a root of a polynomial is obtained when the width of the box is lower than `Accuracy_Variable` and the width of the evaluation of the polynomial is lower than `Accuracy`. If the confidence in the routine that solve numerically a polynomial is low the best choice is 1 otherwise the best choice is 3
- `rand`: every `rand` iteration the algorithm will consider that the current box is the one in the list that has the largest width. Such random permutation may allow to determine the minimal and maximal real root more quickly. This number must be neither too low (otherwise the maximal memory available may be exceeded) nor too large (otherwise the algorithm may focus on some part of the search space while the optimum is located in another part). A good compromise is 100.
- `M`: the maximum number of boxes which may be stored. See the note 2.3.4.5

- `accuracy_Variable`: the maximal width of the range of the polynomial unknown to be a solution, see the note 2.3.4.6
- `Accuracy`: the maximal width of the polynomial evaluation for a solution, see the note 2.3.4.6
- `AccuracyM`: the accuracy with which the optimum is determined. The absolute value of the difference between the real optimum and the calculated one should not exceed this value
- `Lowest_Root`, `Highest_Root`: point interval giving the minimal and maximal real root
- `Place`: the value of the parameters for which the optimum is obtained: the first line is for the minimum and the second line for the maximum
- `Stop`, `Seuil`: if `Stop` is set to 1
  - when looking for a minimum only the procedure exit as soon as a minimum lower than `Seuil[0]` has been found
  - when looking for a maximum only the procedure exit as soon as a maximum greater than `Seuil[1]` has been found
  - when looking both for a minimum and a maximum the procedure exit as a minimum lower than `Seuil[0]` OR a maximum greater than `Seuil[1]` has been found

  If `Stop` is set to 2 when looking both for a minimum and a maximum the procedure exit as a minimum lower than `Seuil[0]` AND a maximum greater than `Seuil[1]` has been found (otherwise as the same behavior than 1)
- `Solve_Poly`: a procedure that compute the real roots of a polynomial. It takes as argument the coefficients of the polynomial, a pointer to an integer that is initially the degree of the polynomial and the real roots are stored in the last argument. This procedure returns the number of real roots or -1 if the computation has failed. ALIAS provides as possible procedure `ALIAS_Solve_Poly`.
- `Simp_Proc`: a user-supplied procedure that take as input the current box and may proceed to some reduction of the width of the box or even determine that there is no solution for this box, in which case it should return -1.

The confidence in this procedure is at the same level than the confidence in the numerical algorithm that solve a polynomial.

The return code is:

- 1: algorithm has succeeded
- 0: result is not guaranteed
- -1: algorithm has failed, not enough memory
- -2: largest root of the polynomial lower than the given lower bound
- -3: smallest root of the polynomial larger than the given upper bound
- -4: error in the type of the equations
- -5: error, more than one function to optimize
- -100: in the mixed bisection mode the number of variables that will be bisected is larger than the number of unknowns
- -150: `ALIAS_Delta3B` or `ALIAS_Max3B` have not the right dimension `Nb_Parameter`+1
- -200: one of the value of `ALIAS_Delta3B` or `ALIAS_Max3B` is negative or 0
- -300: one of the value of `ALIAS_SubEq3B` is not 0 or 1

- -1000: `Single_Bisection` has an incorrect value
- -1500: `Degree` is lower than 0
- -2000: `Nb_Parameter` is lower or equal to 0
- -3000: we use the full bisection mode and the problem has more than 10 unknowns
- -3500: `Nb_Constraints` is lower than 0
- -4000: `Type` not between 0 and 2
- -4500: `Stop_First_Sol` not between 0 and 2
- -5000: `Use_Solve` not between 0 and 4
- -6000: `Place` has not 2 rows or `Nb_Parameter`+1 columns
- -6500: the initial estimate have incompatible lower and upper bound

The possible bisection mode are:

- 1: if the polynomial parameter has a value that is better than the current optimum, then this variable is bisected otherwise mode 1 of section 2.4.1.3
- 2: mode 1 of section 2.4.1.3 if the gradient is not available
- 3,4: mode 1 of section 2.4.1.3
- 5: mode 5 of section 2.4.1.3

For mode 2 if the gradient is available and if the polynomial parameter has a value that is better than the current optimum, then this variable is bisected otherwise we use the smear function to determine the bisected variable.

6.2 Possible parameters values for a given range on the real roots

It may be of interest to determine what may be the possible values of the parameters of a parametric polynomial such that the real roots of the corresponding polynomial are all enclosed in a given interval. `ALIAS` provides two routines for that purpose.

6.2.1 Approximation of the set of solutions

Let consider the *parameters space* i.e. a n dimensional space where each of the dimension corresponds to one of the n parameters. A point in this space corresponds to a unique value for all the parameters and therefore to a specific polynomial. In the parameters space there are possibly a set $\mathcal{S}$ of regions such that for any point in the region(s) the corresponding polynomial has all its root within the given interval. The purpose of the following procedure is to determine an approximation of $\mathcal{S}$. This approximation $\mathcal{A}$ will be constituted of a set of n dimensional boxes which are guaranteed to be included in $\mathcal{S}$ and that will be written in a file. During the calculation the boxes whose width is lower than a given threshold ϵ and for which the algorithm has been unable to determine if they are fully enclosed in $\mathcal{S}$ will be neglected. A possible index for measuring the quality of the approximation $\mathcal{A}$ is the ratio V/V_n between the total volume V of the boxes written into the file over the total volume V_n of the boxes that have been neglected as the volume of $\mathcal{S}$ is lower or equal to $V+V_n$.

The procedure is:

```
int ALIAS_Min_Max_EigenValues_Area(int Degree,int Nb_Parameter,
          int Has_Interval,
          INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* TheCoeffCentered)(INTERVAL_VECTOR &,double),
          int Nb_Constraints,INTEGER_VECTOR &Type_Eq,
```

```
int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
int Has_Matrix,
INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
int Has_Gradient,
INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
INTERVAL & TheDomain,INTERVAL_VECTOR & TheDomain_Parameter,
int Nb_Points,int Use_Solve,int rand,int Strong,int Iteration,
double Accuracy_Variable,double Accuracy,double AccuracyM,double AccuracyB,
double *Volume_Result,double *Volume_Neglected,double Seuil,
char *FileName,int Has_Input,char *File_Input,
int (* Solve_Poly)(double *, int *,double *),int RealRoot,
INTERVAL_VECTOR (* Evaluate_Complex)(int,int,INTERVAL_VECTOR &),
int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where the arguments are similar to the one of the previous procedure except for:

- `TheCoeffCentered`: a procedure that returns the coefficients in x of the polynomial P(x+a). Takes as argument the parameters vector and a
- `AccuracyB`: the threshold ϵ for the maximal width of the neglected boxes
- `Volume_Result`: the total volume of the boxes that have been determined to be enclosed in the regions
- `Volume_Neglected`: the total volume of the boxes that have been neglected
- `Seuil`: the interval [`Seuil[0], Seuil[1]`] defines the allowed range for the real roots of the polynomial
- `FileName`: the name of the file in which will be written the boxes that are included in the regions
- `Has_Input`, `File_Input`: the purpose of these variables is to allow an incremental improvement of the approximation. Indeed after a first run with a given ϵ the quality index may be not satisfactory. It is possible to improve it by decreasing the value of ϵ for a second run but it means that the boxes that have been determined to be enclosed in the region during the first run will be considered again, thereby leading to a loss of efficiency. These arguments allow a better control. During the first run if `Has_Input` has been set to 1 the neglected boxes will be stored in the file `File_Input`. During the second run (and the subsequent run if needed) `Has_Input` will be set to 2 and the set of boxes to be considered by the algorithm will be read from the file `File_Input`. During this type of run the neglected boxes will still be written in the file, allowing another run of the algorithm if needed. Hence the total volume of the boxes enclosed in the region will be the sum of the `Volume_Result` while the total volume of the neglected boxes will be the obtained during the last run of the algorithm. If `Has_Input` is set to 3 the neglected boxes will not be saved in a file.
- `Strong`: if 1 we use a secondary algorithm to determine if for a given box all the polynomial roots are in the range
- `RealRoot`: 0 if we are considering only real roots, 1 for the real part of the roots, 2 if we consider polynomial whose roots are all real and 3 if we consider polynomial with at least one real root
- `Evaluate_Complex(i1,i2,X`: let P be the polynomial and U=P(Seuil_First_Sol[0]+I b), V=P(Seuil_First_Sol[1]+I b). Let a_1 be the real part of U, a_2 the complex part of U, a_3 the real part of V and a_4 the complex part of V. The procedure will return in its interval vector the value of a_{i1} to a_{i2}. `X` is a `Nb_Parameter`+1 interval vector, the last one being the value of b

This procedure returns the number of boxes written in the result file or a negative number if the calculation has failed. The possible negative return code are:

- `-2`, `-3`: errors on the bounds for the roots
- `-10`: `Iteration` is lower than 10
- other values: the procedure that compute the minima and maximal real rots in a box has failed

6.2.2 Largest square enclosed in the regions

This second procedure allows to compute the largest square (up to a pre-defined accuracy α) that is enclosed in the region. This largest box can clearly be obtained from the result of the previous algorithm but this weaker procedure will be faster.

The principle is to have a set of boxes whose first element is a range for the center of the box and second element is a possible value for the length of the half-edge of the square. The main algorithm will test if for a given box the center may be a candidate to be the center of a square of half-edge $R+\alpha$ (where R is the current optimum) using some heuristics and if the answer is positive will compute the minimal and maximal root of the polynomials defined by this square using a secondary algorithm. If these values are compatible with the bound the current optimum will be updated.

```
int ALIAS_Geometry_Carre(int Degree,int Nb_Parameter,
            INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
            int Nb_Constraints,INTEGER_VECTOR &Type_Eq,INTEGER_VECTOR &Imperatif,
            int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
            int Has_Matrix,
            INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
            int *Has_Gradient,
            INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
            INTERVAL & TheDomain,INTERVAL_VECTOR & TheDomain_Parameter,
            int Nb_Points,int Use_Solve,int rand,
            int Iteration_Geometry,int Iteration_Polynom,
            double Accuracy_Variable,double Accuracy,double Accuracy_Geometry,
            double Accuracy_Polynom,INTERVAL_VECTOR &Solution,double *Seuil,
            int (* Solve_Poly)(double *, int *,double *),
            int (* Simp_Proc)(INTERVAL_VECTOR &),
            int (* Simp_Proc_Pol)(INTERVAL_VECTOR &))
```

The arguments are the same than for the previous procedure except for:

- `Imperatif`: an array of integer. If there are constraints that are imperative, meaning that the polynomial cannot be evaluated if they are not satisfied (for example the constraints is that some term is positive as in the coefficients of the polynomial this term appears as a square root) you may set the corresponding integer to 1. If this array has dimension 0 all the constraint will supposed to be not imperative
- `Iteration_Geometry`: the number of boxes that may be used by the main algorithm
- `Iteration_Polynom`: the number of boxes that may be used by the secondary algorithm
- `Accuracy_Geometry`: the value of α
- `Accuracy_Polynom`: the accuracy used for the secondary algorithm. Note that this parameter play a role not only on the computation time but also on the bound for the root. Indeed the algorithm will verify that in the square there is no polynomial with a root lower than `Seuil[0]+Accuracy_Polynom` or larger than `Seuil[1]-Accuracy_Polynom`
- `Solution`: the point interval is the center of the largest square while the second is the value of the half-edge
- `Simp_Proc`: a simplification procedure used only in the main algorithm. The flag `ALIAS_Simp_Main` is set to 1 when this procedure is called right after the bisection process.
- `Simp_Proc_Polynom`: a simplification procedure used only in the secondary algorithm

This procedure will return:

- -1000: error in the `Single_Bisection` flag that should be between 0 and 5
- -4: error in `Type_Eq`

- -3: the lowest root of all the polynomials is greater than `Seuil[1]`
- -2: the highest root of all the polynomials is lower than `Seuil[0]`
- -1: the algorithm has failed
- 0: the result is not guaranteed
- 1:the result is guaranteed

During the calculation the flag `ALIAS_Has_OptimumG` will be set to 1 as soon as an optimum is found: the center of the current optimal geometry is the mid point of the interval vector `ALIAS_Vector_OptimumG` while is edge is in the interval `ALIAS_OptimumG`.

The secondary algorithm uses the flag `Single_BisectionG` and `Reverse_StrorageG` that play the same role than `Single_Bisection` and `Reverse_Storage` in the general solving algorithms (see sections 2.3.1.3 and 2.3.1.2).

6.3 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:

```
int ALIAS_Min_Max_CN(int Degree,
     int Nb_Parameter,INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
     int Nb_Constraints,INTEGER_VECTOR &Type_Eq,
     int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
     int Has_Matrix,
     INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
     int Has_Gradient,
     INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
     INTERVAL & TheDomain,INTERVAL_VECTOR & TheDomain_Parameter, int Type,
     int Nb_Points,int Absolute,
     int rand,int Iteration,
     double Accuracy_Variable,double Accuracy,double AccuracyM,
     INTERVAL &Lowest,INTERVAL &Highest,
     INTERVAL_MATRIX &Place,int  Stop, double *Seuil,
     int (* Solve_Poly)(double *, int *,double *),
     int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where the arguments are identical than for the previous procedure except for:

- `Absolute`: 0 if looking for the ratio minimal root over maximal root, 1 if looking for the ratio in absolute value
- `AccuracyM`: the accuracy with which the ratio will be computed
- `Lowest, Highest`: minimal and maximal value of the condition number

This procedure allows for the calculation of the minimal and maximal condition number of a matrix. Various bisection methods are available through the use of the integer `Single_Bisection`:

- 1: mode 1 of section 2.4.1.3
- 2: mode 6 of section 2.4.1.3 if the gradient is not available
- 3,4: mode 1 of section 2.4.1.3
- 5: mode 5 of section 2.4.1.3

A specific procedure is used when the gradient is available. If the interval for the root includes 0 we bisect it. Otherwise we use the smear function to decide which other variable should be bisected.

6.4 Kharitonov polynomials

Kharitonov polynomials are special polynomials that have constant values for their coefficients and are associated to a parametric polynomial. 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 roots of the same sign. For a polynomial $P = \sum C(i)x^{i-1}$ the four Kharitonov polynomials are:

$$P -- = Inf(C1) + Inf(C2)x + Sup(C3)x^2 + Sup(C4)x^3 + InfInfSupSup$$
$$P -+ = Inf(C1) + Sup(C2)x + Sup(C3)x^2 + Inf(C4)x^3 + InfSupSupInf$$
$$P +- = Sup(C1) + Inf(C2)x + Inf(C3)x^2 + Sup(C4)x^3 + SupInfInfSup$$
$$P ++ = Sup(C1) + Sup(C2)x + Inf(C3)x^2 + Inf(C4)x^3 + SupSupInfInf$$

6.4.1 Implementation

The following procedure allows to determine if a polynomial has a real root within a given range:

```
int Kharitonov(int Degree,
       INTERVAL_VECTOR (* TheCoeff)(double a,INTERVAL_VECTOR &),
       int (* Solve_Poly)(double *, int *,double *),
       INTERVAL_VECTOR &Input)
```

where:

- `Degree`: the degree of the polynomial
- `TheCoeff`: a procedure that allows to compute the interval coefficients of the polynomial at a point `a` (i.e. the coefficients of the polynomial $P(x-a)$)
- `Solve_Poly`: a procedure that compute the real part of the roots of a polynomial. It takes as argument the coefficients of the polynomial, a pointer to an integer that is initially the degree of the polynomial and the real roots are stored in the last argument. This procedure returns the number of roots or `-1` if the computation has failed. ALIAS provides as possible procedure `ALIAS_Solve_Poly_PR`.
- `Input`: the range for the root

Note that there exists an `ALIAS`-Maple procedure that uses this procedure to generate a simplification procedure, see the ALIAS-Maple manual.

6.5 Gerschgorin circles

6.5.1 Mathematical background

Let $A = ((a_{ij}))$ and the Gerschgorin circles defined as the set of z such that:

$$C_i = |z - a_{ii}| \leq \sum_j |a_{ji}|, j \neq i$$

The roots of the characteristic polynomial are enclosed in the union of the C_i. Furthermore if a circle has no intersection with the other circles, then this circle contains one root of the polynomial. This allow for a fast determination of bounds for the real roots of the polynomial.

6.5.2 Implementation

It is possible to get these bounds by using the procedure:

```
int Gerschgorin(INTERVAL_MATRIX &A,int Size, int Type, INTERVAL &Bound)
```

where:

- `A`: the matrix
- `Size`: the dimension of the matrix
- `Type`: 0 if the matrix is not available, 2 if the matrix is symmetrical, 1 otherwise
- `Bound`: all the roots will be enclosed in this interval

the procedure returns 1 if it has been possible to compute `Bound`.

A more complete procedure allows to get all the Gerschgorin circles and eventually to adjust an interval that is supposed to contain the roots of the polynomial:

```
int Gerschgorin_Simplification(INTERVAL_MATRIX &A,int Size,int Type,
                        INTERVAL &Input,INTERVAL_VECTOR &Circle)
```

The arguments are the same than the previous procedure except for `Input` which is the interval supposed to contain roots of the polynomial and `Circle` which contain the projection of the Gerschgorin circle on the real axis. This procedure returns:

- -1: `Input` does not contain a root of the polynomial
- 0: no change in `Input`
- ≥ 1: `Input` has been improved and the return value gives the number of distinct circles

6.6 Cassini ovals

6.6.1 Mathematical background

The Cassini ovals are another method to determine a bound for the eigenvalues of a matrix. Let $A = ((a_{ij}))$ and the Cassini ovals defined as the set of z such that:

$$O_{jk} = |z - a_{jj}||z - akk \leq (\sum_i |a_{ji}|, i \neq j)(\sum_i |a_{ki}|, i \neq k)$$

Column based Cassini ovals may also be defined. The roots of the characteristic polynomial are enclosed in the union of the row-based and column-based O_{jk}. Although more complicate to calculate the bounds obtained with the Cassini ovals are usually tighter than the bounds obtained with the Gerschgorin circles.

6.6.2 Implementation

The Cassini bounds may be obtained with the procedure:

```
int Cassini_Simplification(INTERVAL_MATRIX &A,int Size,
                  int Type_Matrix, INTERVAL &Input)
```

where

- `A`: the matrix
- `Size`: the dimension of the matrix
- `Type`: 0 if the matrix is not available, 2 if the matrix is symmetrical, 1 otherwise

- `Input`: an estimation of the bounds of the roots

The procedure will return -1 if there is no intersection between `Input` and the Cassini bounds, 1 if there `Input` has been improved, 0 otherwise.

This procedure may also be called with:

```
int Cassini_Simplification(int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
      INTERVAL_VECTOR &Param,int Size, int Type_Matrix, INTERVAL &Input)
```

where `TheMatrix` is a procedure that compute the elements of the matrix according to the interval value of the parameters `Param`.

6.7 Routh

The Routh algorithm allows to calculate the number of roots with positive real part of a polynomial being given the coefficients of the polynomial. It is implemented as:

```
int Routh(int Degree,double *Coeff)
```

that returns the number of roots with positive real part or -1 if it has not been possible to compute the Routh table (because the first element of a row of the Routh table is close to 0).

A similar algorithm allows to deal with polynomial with interval coefficients:

```
INTERVAL Routh(int Degree,INTERVAL_VECTOR &Coeff)
```

This algorithm returns in its interval:

-1,-1 : the Routh table cannot be computed

a,a+1 : there is at least a roots with positive real part but the exact number of roots with positive real part cannot be calculated

a,a : there is exactly a roots with positive real part

A similar procedure may be used when the coefficients are functions of parameters:

```
INTERVAL Routh(int Degree,INTERVAL_VECTOR (* TheCoeff)(int,int,INTERVAL_VECTOR &),
               INTERVAL_VECTOR &Input)
```

where:

- `TheCoeff`: a procedure that allow to calculate the coefficients being given the range for the parameters. It returns a `Degree`+1 interval vector. The two first integers l1, l2 of the procedure allows one to specify which coefficients are calculated. For example if `V=TheCoeff(1,5,Input)`, then `V(1..5)` will be the first 5 coefficients of the polynomial and if `V=TheCoeff(1,Degree+1,Input)`, then all the coefficients of the polynomial will be available in `V`
- `Input`: the intervals for the parameter

If the derivatives of the coefficients with respect to the parameters are known an even better procedure will be:

```
INTERVAL Routh(int Degree,INTERVAL_VECTOR (*TheCoeff)(int,int,INTERVAL_VECTOR &),
               INTERVAL_MATRIX (* TheCoeffG)(int,int,INTERVAL_VECTOR &),
               INTERVAL_VECTOR &Input)
```

where `TheCoeffG` is a procedure that allow to compute the derivatives of the coefficients with respect to the parameters (see note 2.4.2.2). This procedure allows to a certain amount to take into account the dependency between the coefficients.

Note also that the procedure `Routh` of ALIAS-Maple allows an even better calculation of the Routh table when dealing with parametric polynomials as the elements of the Routh table are calculated symbolically.

6.8 Weyl filter

6.8.1 Mathematical background

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 [1]. This procedure may be used for univariate polynomial, with polynomial with interval coefficients or with parametric poynomial. For example it is very efficient for solging the Wilkinson polynomial.

6.8.2 Implementation

The Weyl filter is implemented to manage univariate polynomial (with numerical or interval coefficients) or parametric polynomial. A basic tool of this filter is

```
int Weyl_Filter_Utility(int Degree, double maxroot,
                        INTERVAL_VECTOR &b,INTERVAL_VECTOR &Input)
```

where `Input(1)` is the range for the polynom variable and `b` the b_i coefficients. A more general implementation is

```
int Weyl_Filter(int Degree, INTERVAL_VECTOR &Coeff,double maxroot,
                int (* GetB)(int,INTERVAL_VECTOR &,INTERVAL &,INTERVAL_VECTOR &),
INTERVAL_VECTOR &Input)
```

Here `GetB` is a procedure that calculates the b_i coefficients. Its arguments are the degree of the polynomial, its coefficients, z_0 and the b_i coefficients. It should return 1 if the calculation has been successfull, 0 otherwise. An example of such procedure is `Derive_Polynomial_Expansion`. `Degree` is the degree of the polynomial, `Coeff` its coefficients and `Input(1)` is the range for the polynomial unknown.

This procedure has several variants:

```
int Weyl_Filter(int Degree, INTERVAL_VECTOR &Coeff,double max_root,
                INTERVAL_VECTOR &Input)
```

here the `GetB` procedure will be `Derive_Polynomial_Expansion`. In

```
int Weyl_Filter(int Degree, INTERVAL_VECTOR &Coeff,
                INTERVAL_VECTOR &Input)
```

here the `GetB` procedure will be `Derive_Polynomial_Expansion` and `maxroot` will be computed by an `ALIAS` procedure. The same procedure will be used in

```
int Weyl_Filter(int Degree, INTERVAL_VECTOR &Coeff,
                int (* GetB)(int,INTERVAL_VECTOR &,INTERVAL &,INTERVAL_VECTOR &),
                INTERVAL_VECTOR &Input)
```

6.9 Coefficient of the characteristic polynomial

In some cases it may be difficult to obtain the analytical form of the coefficients of the characteristic polynomial of a matrix. The following procedure allows to compute these coefficients even if the matrix is an interval matrix:

```
INTERVAL_VECTOR Coeff_CharPoly(int Size,INTERVAL_MATRIX &A)
```

where `Size` is the dimension of the matrix and `A` the matrix. This procedures returns the coefficients in an interval vector C and the polynomial is $\sum C(i)x^{i-1}$.

Chapter 7

Linear algebra

7.1 Calculating determinant

7.1.1 Scalar and interval matrix

The determinant of a scalar matrix may be calculated with the procedure `ALIAS_MRINVD` that is also used to determine the inverse of the matrix, see section 7.2.

The determinant of an interval matrix may be calculated by using Gaussian elimination with the procedure

```
int ALIAS_Det_By_Gaussian_Elim(INTERVAL_MATRIX &B, INTERVAL &DET)
```

This procedures returns 1 if the determinant has been successfully calculated and is returned in `DET`, 0 otherwise.

There are five main procedures to compute an interval evaluation of the determinant of an interval square matrix:

```
INTERVAL Slow_Determinant(INTERVAL_MATRIX &A)
INTERVAL Slow_NonZero_Determinant(INTERVAL_MATRIX &A)
INTERVAL Medium_Determinant(INTERVAL_MATRIX &A)
INTERVAL Fast_Determinant(INTERVAL_MATRIX &A)
INTERVAL VeryFast_Determinant(INTERVAL_MATRIX &A)
```

These procedures differ only by the computation time (large for the Slow procedure as soon as the dimension of the matrix is larger than 5) and the accuracy of the interval evaluation (which is the worst for the Fast procedure). The `Medium_Determinant` procedure is a good compromise between efficiency and speed. The procedure `Slow_NonZero_Determinant` is a special occurrence of the procedure `Slow_Determinant` which will return as soon it has determined that the interval evaluation of the determinant does not include 0. Note that if the flag `ALIAS_Use_Gaussian_Elim_In_Det` is set to 1 (the default value being 0), then these procedures will also use the Gaussian elimination to improve the evaluation of the determinant.

There are also special version of the previous procedures:

```
INTERVAL Slow_Determinant22(INTERVAL_MATRIX &A,
              INTERVAL (* TheDet22)(INTEGER_VECTOR &,INTEGER_VECTOR &,INTERVAL_VECTOR &),
              INTERVAL_VECTOR &Input)
INTERVAL Medium_Determinant22(INTERVAL_MATRIX &A,
             INTERVAL (* TheDet22)(INTEGER_VECTOR &,INTEGER_VECTOR &,INTERVAL_VECTOR &),
             INTERVAL_VECTOR &Input)
INTERVAL Fast_Determinant22(INTERVAL_MATRIX &A,
              INTERVAL (* TheDet22)(INTEGER_VECTOR &,INTEGER_VECTOR &,INTERVAL_VECTOR &),
              INTERVAL_VECTOR &Input)
```

They differ because the calculation of all the dimensions 2 minors are computed using the `TheDet22` procedure. The syntax of this procedure is:

```
INTERVAL TheDet22(INTEGER_VECTOR &ROW,INTEGER_VECTOR &COL,INTERVAL_VECTOR &Input)
```

The integer vectors ROW and COL will have all components equal to 0 except for 2 element, which indicate which rows and columns are used to produce the 2×2 determinant. A dimension 2 matrix will be obtained from the initial matrix by removing all the rows and columns whose index in the integer vectors are 0. The procedure must return an interval corresponding to the determinant of this dimension 2 matrix, the unknowns having for range the value indicated by Input.

A more general implementation of the previous procedure is

```
INTERVAL Determinant22(INTERVAL_MATRIX &A,int Minor,int Row,
     INTERVAL (* TheDet22)(INTEGER_VECTOR &,INTEGER_VECTOR &,INTERVAL_VECTOR &),
     INTERVAL_VECTOR &Input)
```

In this procedure Minor indicates the size of the minor that can be computed using the procedure TheDet22 and Row indicates if an expansion by row (1) or by column (0) is used.

There are also procedures to compute the derivatives of a determinant

```
INTERVAL Fast_Derivative_Determinant(INTERVAL_MATRIX &A,INTERVAL_MATRIX &AG);
INTERVAL Medium_Derivative_Determinant(INTERVAL_MATRIX &A,INTERVAL_MATRIX &AG);
INTERVAL Slow_Derivative_Determinant(INTERVAL_MATRIX &A,INTERVAL_MATRIX &AG);
```

A is a matrix that contain the interval evaluation of the component of the matrix and AG is a matrix that contain the interval evaluation of the derivatives of the components of the matrix. In other words if $M = ((a_{ij}))$ then A $= (([\underline{a_{ij}}, \overline{a_{ij}}]))$ and if AG $= (([\underline{da_{ij}/dx}, \overline{da_{ij}/dx}]))$, then the procedure will return the interval evaluation of $d|M|/dx$.

If a procedure TheDet22 for calculating the minor is available you may use

```
INTERVAL Derivative_Determinant22(INTERVAL_MATRIX &A,INTERVAL_MATRIX &AG,
          int Minor,
          INTERVAL (* TheDet22)(INTEGER_VECTOR &,INTEGER_VECTOR &,INTERVAL_VECTOR &),
          INTERVAL_VECTOR &Input)
```

There are also procedures to compute the second order derivatives of a determinant

```
INTERVAL Fast_Hessian_Determinant(INTERVAL_MATRIX &A,INTERVAL_MATRIX &AG, INTERVAL_MATRIX &AH)
INTERVAL Medium_Hessian_Determinant(INTERVAL_MATRIX &A,INTERVAL_MATRIX &AG, INTERVAL_MATRIX &AH)
INTERVAL Slow_Hessian_Determinant(INTERVAL_MATRIX &A,INTERVAL_MATRIX &AG, INTERVAL_MATRIX &AH)
```

For a matrix $M = ((a_{ij}))$ then A $= (([\underline{a_{ij}}, \overline{a_{ij}}]))$ and if AG $= (([\underline{da_{ij}/dx}, \overline{da_{ij}/dx}]))$ and AH $= (([\underline{d^2a_{ij}/dx\ dy}, \overline{d^2a_{ij}/dx\ dy}]))$ then the procedure will return the interval evaluation of $d^2|M|/dx\ dy$.

You may also obtain an upper bound for the determinant of a matrix by using the procedure:

```
double Hadamard_Determinant(MATRIX &J)
```

where we use the Hadamard bound of the determinant D of a $n \times n$ matrix J

$$D \leq \prod_{i=1}^{i=n} \sqrt{\sum_{j=1}^{j=n} J_{ij}^2}$$

A similar procedure exists for an interval matrix

```
INTERVAL Hadamard_Determinant(INTERVAL_MATRIX &J)
```

The determinant of a scalar matrix may be calculated with

```
double Determinant(MATRIX &J)
```

7.1.2 Polynomial matrix

A polynomial matrix is assumed is assumed here to be a matrix whose elements are univariate polynomial in the same variable. The determinant of such matrix (which is a polynomial) may be computed with the procedure

```
int Determinant_Characteristic(POLYNOMIAL_MATRIX A,
                  INTERVAL_VECTOR &Coeff)
```

where

- `A`: a `POLYNOMIAL_MATRIX` structure which describes the polynomial matrix. This structure is composed of
  - `dim`: the dimension of the matrix
  - `Coeff`: an interval vector with indicates the coefficients of the polynomial in the matrix, row by row
  - `Order`: an integer matrix. `m=Order(i,j)` indicates that the element at the i-th row and j-th column has coefficients `Coeff(l)` to `Coeff(m)` where `l` is `Coeff(i,j-1)`+1 if `j` is greater than 1, `Coeff(i-1,dim)` otherwise. Hence `Order(1,1)=3` indicate that the first elements has as coefficient `Coeff(1), Coeff(2), Coeff(3)` and hence is a second order polynomial
- `Coeff`: the coefficients of the determinant polynomial

Note that this procedure is safe: even for a scalar matrix the coefficients of the determinant will always include the coefficient of the exact determinant.

7.2 Matrix inverse

The inverse of a scalar matrix may be used by using the `Inverse` function of the `BIAS/Profil` package or with

```
void ALIAS_MRINVD(VECTOR &A,VECTOR &B,int N,int *KOD,double *DET,double EPS,
                  INTEGER_VECTOR &IL,INTEGER_VECTOR &IC)
```

where

- `A` is the matrix given by column
- `B` is the inverse of `A`
- `N` is the dimension of the matrix
- `KOD` is a return code. If `KOD` is 0 then `A` is estimated not to have an inverse, otherwise `KOD` is set to 1
- `DET`: the determinant of `A`
- `EPS`: a threshold, if a pivot has an abolute value less than this value, then it is assumed to be 0
- `IL, IC`: working table of size `N`

The inverse of an interval matrix may be defined as the set of matrices corresponding to the inverse of a matrix included in the set defined by the interval matrix. This set cannot usually be computed exactly but a set of matrices guaranteed to include the inverse interval matrix may be computed. The following procedure allows one to compute such overestimation.

```
int Inverse_Interval_Matrix(int Dim,int cond,INTERVAL_MATRIX &Jac,INTERVAL_MATRIX &InvJac)
```

where `Dim` is the dimension of the matrix `Jac`. The flag `cond` has to be set to 1 if pre-conditioning is used, 0 otherwise. Pre-conditionning will usually leads to a smaller overestimation but is more computer intensive. This procedures returns 1 if it has been able to compute the inverse, 0 otherwise.

Note that using this procedure should not be used for solving an interval linear system.

7.3 Solving systems of linear equations

7.3.1 Mathematical background

Let us assume that we have an $n \times n$ interval linear system:

$$\mathbf{A}(\mathbf{X}).Y = b(\mathbf{X}) \tag{7.1}$$

where the $\mathbf{A}, b$ elements are functions of the unknowns $\mathbf{X}$. The above equation describes a family of linear system and the *enclosure* of this interval linear system is a box that enclose the solutions for Y of each member of the family of linear systems.

A classical scheme for finding the enclosure is to use an interval variant of the *Gauss elimination* scheme [18].

When the unknowns lie in given ranges we may compute an interval evaluation $\mathbf{A}^{(0)}$ of $\mathbf{A}$ and an interval evaluation $b^{(0)}$ of b. The Gauss elimination scheme may be written as [18]

$$A_{ik}^{(j)} = A_{ik}^{(j-1)} - A_{ij}^{(j-1)} A_{jk}^{(j-1)} / A_{jj}^{(j-1)} \quad \forall\ i \text{ with } j > k \tag{7.2}$$

$$b_i^{(j)} = b_i^{(j-1)} - A_{ij}^{(j-1)} b_j^{(j-1)} / A_{jj}^{(j-1)} \tag{7.3}$$

The enclosure of the variable Y_j can then be obtained from $Y_{j+1}, \ldots, Y_n$ by

$$Y_j = (b_j^{(j-1)} - \sum_{k>j} A_{jk}^{(j-1)} Y_k) / A_{jj}^{(j-1)} \tag{7.4}$$

Note that the elements at iteration j can be computed only if the interval $A_{jj}^{(j-1)}$ does not include 0.

A drawback of this scheme is that the family of linear systems that will be obtained for all instances of $\mathbf{X}$ is usually a subset of the family of linear systems defined by $\mathbf{A}^{(0)}$, $b^{(0)}$, as we do not take into account the dependency of the elements of $\mathbf{A}, b$. Furthermore the calculation in the scheme involves products, sums and ratio of elements of $\mathbf{A}, b$ and their direct interval evaluation again do not take into account the dependency between the elements. Hence a direct application of the Gauss scheme will usually lead to an overestimation of the enclosure.

A possible method to reduce this overestimation is to consider the system

$$J\mathbf{A}(\mathbf{X}).Y = Jb(\mathbf{X})$$

where J is an arbitrary $n \times n$ matrix. The above system has the same solutions than the system (7.1) but for some matrix J the enclosure of the above interval system may be included in the enclosure obtained by the Gauss elimination scheme on the initial system: this is called a *pre-conditioning* of the initial system. It is usually considered that a good candidate for J is the matrix whose elements are the mid-point of the ranges for the elements of $\mathbf{A}$.

But even with a possible good J the dependency between the elements of $\mathbf{A}, b$ are not taken into account and hence the overestimation of the enclosure may be large.

A first possible way to reduce this overestimation is to improve the interval evaluation of $\mathbf{A}^{(0)}$, $b^{(0)}$ by using the derivatives of their elements with respect to $\mathbf{X}$ and the procedure described in section 2.4.2.3. Note that the procedures necessary to compute the elements of $\mathbf{A}^{(0)}$, $b^{(0)}$ and their derivatives may be obtained by using the `MakeF, MakeJ` procedures of ALIAS-maple.

But to improve the efficiency of the procedure it must be noticed that at iteration j an interval evaluation of the derivatives of A_{ik}, b_i with respect to $\mathbf{X}$ may be deduced for the derivatives of the elements computed at iteration $j-1$. As we have the derivatives of the elements at iteration 0 we may then deduce the derivatives of the elements at any iteration and use these derivatives to improve the interval evaluation of these elements (see sections 2.4.1.1 and 2.4.2.3).

7.3.2 Implementation

The simplest Gaussian elimination scheme is implemented as:

```
int Gauss_Elimination(INTERVAL_MATRIX &Ain,
                      INTERVAL_VECTOR &b,INTERVAL_VECTOR &bout)
```

where

- `Ain`: the **A** interval matrix
- `b`: the **b** interval vector
- `bout`: the enclosure of the set of solutions

The procedure returns 1 if it has succeeded in finding the enclosure, 0 otherwise (one of the interval pivot in the Gaussian scheme includes 0).

This computation for the initial system and a pre-conditioned system has been implemented in the procedure:

```
int Gauss_Elimination_Derivative(MATRIX &Cond,INTERVAL_MATRIX &Ain,
              INTERVAL_MATRIX &ACondin,
               const INTERVAL_VECTOR bin,
               const INTERVAL_VECTOR bCondin,
              INTERVAL_VECTOR &bout,
              INTERVAL_VECTOR & Param,
              INTERVAL_VECTOR (* Func)(int l1, int l2, INTERVAL_VECTOR & v_IS),
              INTERVAL_MATRIX (* JFunc)(int l1, int l2, INTERVAL_VECTOR & v_IS),
              INTERVAL_VECTOR (* bFunc)(int l1, int l2, INTERVAL_VECTOR & v_IS),
              INTERVAL_MATRIX (* JbFunc)(int l1, int l2, INTERVAL_VECTOR & v_IS))
```

where

- `Cond`: the pre-conditioning matrix. If all the elements of `Cond` are 0 the procedure will implement the Gauss elimination scheme only on the initial system
- `Ain`: the interval evaluation of the **A** matrix
- `ACondin`: the interval evaluation of the `Cond`**A** matrix
- `bin`: the interval evaluation of b
- `bCondin`: the interval evaluation of `Cond`b
- `bout`: the enclosure of the interval linear system
- `Param`: the ranges for **X**
- `Func`: a procedure that computes the interval evaluation of the elements of **A**. These elements are stored rows by rows in an $n \times n$ interval vector (see the note 2.3.4.3)
- `Jfunc`: a procedure that computes the interval evaluation of the derivatives of the elements of **A**, see the note 2.4.2.2
- `bFunc`:a procedure that computes the interval evaluation of the elements of b, see the note 2.3.4.3
- `Jbfunc`: a procedure that computes the interval evaluation of the derivatives of the elements of b, see the note 2.4.2.2

This procedure will return 1 if the Gauss elimination scheme has been completed, 0 otherwise. It will return in general a much more better enclosure than the classical interval Gauss elimination scheme.

For example consider the system

$$\begin{bmatrix} x & y \\ x & x \end{bmatrix} Y = \begin{bmatrix} x \\ x \end{bmatrix}$$

for x in [3,4] and y in [1,2]. Using the classical Gauss elimination scheme the enclosure of the solution is:

$$Y_1 = [0.76923, 10] \quad Y_2 = [-13, -0.0769]$$

while if we use the derivatives we get

$$Y_1 = [0.9166, 2.6666] \quad Y_2 = [-2, -.66666]$$

while the solutions are $Y_1 = (y + x)/x$ (which has an enclosure of [1.25,1.666]), $Y_2 = -1$.

The ALIAS-Maple procedure `LinearBound` implements this calculation.

7.4 Regularity of parametric interval matrices

A *parametric interval matrix* is a matrix whose elements are functions of unknowns whose values lie in some pre-defined ranges. It is therefore a matrix that has a higher structure than the classical interval matrix (i.e. matrices whose elements are intervals) as there may be dependency between the elements of the matrix.

A sophisticated procedure to check for the regularity of a parametric matrix is based on the following scheme: the sign of the determinant is calculated for a point value of the input parameters (say positive), then the algorithm try to find another input value such that the sign of the corresponding determinant is negative.

7.4.1 Implementation

```
int Matrix_Is_Regular(int dimA,
      INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
      int (* A_Cond)(int dimA,INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),INTERVAL_VECTOR & v_IS,INTERVAL_MATRIX &K),
      INTERVAL_MATRIX (* A_Cond_Left)(INTERVAL_VECTOR & v_IS_Left),
      INTERVAL_MATRIX (* A_Cond_Right)(INTERVAL_VECTOR & v_IS_Right),
      INTERVAL Determinant_Matrix(INTERVAL_MATRIX &A,INTERVAL_VECTOR &X),
      INTERVAL Determinant_A_Cond_Left(INTERVAL_MATRIX &A,INTERVAL_VECTOR &X),
      INTERVAL Determinant_A_Cond_Right(INTERVAL_MATRIX &A,INTERVAL_VECTOR &X),
      int Type_Cond,
        INTEGER_VECTOR &Type_Determinant,
      int Iteration,
      INTERVAL_VECTOR &X,
      int (* Simp)(int dimA,INTERVAL_MATRIX & Acond,INTERVAL_VECTOR
& v_IS))
```

where

- `dimA`: the dimension of the matrix
- `Func`: a procedure that allows to compute the entries of the matrix, row by row. `Func(l1,l2,X)` computes the element at row l1, column l2 of the matrix for the input `X`
- `A_Cond`: a procedure that compute a conditioning matrix. `K`. It returns 1 if the conditioning matrix has been calculated, 0 otherwise. You may use the built-in procedure `ALIAS_A_Cond_Mid` that calculate the scalar matrix for the mid point of the input parameters and returns its inverse. If you don't intend to use conditioning you may use the dummy procedure `ALIAS_A_Cond_Void` that just returns 0. In that case the procedure that should be used to calculated the conditioned matrix may be `ALIAS_Cond_Void`
- `A_Conf_Left`: a procedure to compute the lefts conditioned matrix `AK`. It takes as input a vector constituted first of the elements of `X` followed by the elements of `K`, arranged row by row. If no such procedure is available the dummy procedure `ALIAS_A_Cond_Void` may be used.
- `A_Conf_Right`: a procedure to compute the lefts conditioned matrix `KA`. It takes as input a vector constituted first of the elements of `X` followed by the elements of `K`, arranged row by row. If no such procedure is available the dummy procedure `ALIAS_A_Cond_Void` may be used.

- `Type_Cond`: indicates how the conditioning is used: 0 if no conditioning is used, 1 if `AK` is being used, 2 if `KA` is used and 3 if both `AK` and `KA` are used
- `Type_Det`: an integer of dimension 3 that indicates which procedure is used to calculate the determinant. The value of the integer is 1 for `Fast_Determinant`, 2 for `Medium_Determinant` and 3 for `Slow_Determinant`. The first integer is for `A`, the second for `AK` and the third for `KA`
- `Iteration`: the maximum number of boxes that will be used by the algorithm
- `X`: the ranges for the input parameters
- `Simp`: a user supplied simplification procedure that returns -1 if all the matrices for the input parameter `X` are regular. If no such procedure is available you may use the dummy procedure `ALIAS_Simp_Matrix_Void`

This procedure returns 1 if all matrices are regular, -1 if a singular matrix has been found, -2 if the algorithm has failed and -3 if the regularity cannot be ascertained as for a scalar input `X` the sign of the determinant cannot be ascertained.

The flag `Simp_In_Cond` may be used to design the simplification procedure. It is set to 0 when the simplification procedure is used to check the initial matrix A, to -1 or -2 when testing the regularity of the conditioning matrix, to 1 when testing the regularity of $\mathbf{AK}$ and to 2 for $\mathbf{KA}$.

This procedure may also be used with the following syntaxes:

```
int Matrix_Is_Regular(int dimA,
    INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
    int (* A_Cond)(int dimA,INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),INTERVAL_VECTOR & v_IS,INTERVAL_MATRIX &A),
    INTERVAL_MATRIX (* A_Cond_Left)(INTERVAL_VECTOR & v_IS_Left),
    INTERVAL_MATRIX (* A_Cond_Right)(INTERVAL_VECTOR & v_IS_Right),
    INTERVAL Determinant_Matrix(INTERVAL_MATRIX &A,INTERVAL_VECTOR &X),
    INTERVAL Determinant_A_Cond_Left(INTERVAL_MATRIX &A,INTERVAL_VECTOR &X),
    INTERVAL Determinant_A_Cond_Right(INTERVAL_MATRIX &A,INTERVAL_VECTOR &X),
    int Type_Cond,
        INTEGER_VECTOR &Type_Determinant,
    int Iteration,
    INTERVAL_VECTOR &Domain)

int Matrix_Is_Regular(int dimA,
    INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
    int (* A_Cond)(int dimA,INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),INTERVAL_VECTOR & v_IS,INTERVAL_MATRIX &A),
    INTERVAL_MATRIX (* A_Cond_Left)(INTERVAL_VECTOR & v_IS_Left),
    INTERVAL_MATRIX (* A_Cond_Right)(INTERVAL_VECTOR & v_IS_Right),
    int Type_Cond,
        INTEGER_VECTOR &Type_Determinant,
    int Iteration,
    INTERVAL_VECTOR &Domain,
    int (* Simp)(int dimA,INTERVAL_MATRIX & Acond,INTERVAL_VECTOR
& v_IS))

int Matrix_Is_Regular(int dimA,
    INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
        int Type_Determinant,
    int Iteration,
    INTERVAL_VECTOR &Domain)

int Matrix_Is_Regular(int dimA,
    INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
        int Type_Determinant,
```

```
    int Iteration,
    INTERVAL_VECTOR &Domain,
    int (* Simp)(int dimA,INTERVAL_MATRIX & Acond,INTERVAL_VECTOR
&v_IS))

int Matrix_Is_Regular(int dimA,
    INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
    int (* A_Cond)(int dimA,INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),INTERVAL_VECTOR & v_IS,INTERVAL_MATRIX &A),
    INTERVAL_MATRIX (* A_Cond_Left)(INTERVAL_VECTOR & v_IS_Left),
    INTERVAL_MATRIX (* A_Cond_Right)(INTERVAL_VECTOR & v_IS_Right),
    int Type_Cond,
        INTEGER_VECTOR &Type_Determinant,
    int Iteration,
    INTERVAL_VECTOR &Domain)

int Matrix_Is_Regular(int dimA,
    INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
    INTERVAL_MATRIX (* A_Cond_Left)(INTERVAL_VECTOR & v_IS_Left),
    INTERVAL_MATRIX (* A_Cond_Right)(INTERVAL_VECTOR & v_IS_Right),
    int Type_Cond,
        INTEGER_VECTOR &Type_Determinant,
    int Iteration,
    INTERVAL_VECTOR &Domain)

int Matrix_Is_Regular(int dimA,
    INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
    INTERVAL_MATRIX (* A_Cond_Left)(INTERVAL_VECTOR & v_IS_Left),
    INTERVAL_MATRIX (* A_Cond_Right)(INTERVAL_VECTOR & v_IS_Right),
    int Type_Cond,
        INTEGER_VECTOR &Type_Determinant,
    int Iteration,
    INTERVAL_VECTOR &Domain,
    int (* Simp)(int dimA,INTERVAL_MATRIX & Acond,INTERVAL_VECTOR
&v_IS))
```

Note that the 3B method is implemented as a built-in procedure called `ALIAS_3B_Regular_Matrix`, but is usually not very efficient.

7.4.2 Rohn simplification procedure

7.4.2.1 Mathematical background

Let H be the set of vector with components either -1 or 1 and the scalar matrices $\mathbf{A^{uv}}$, $\mathbf{u}, \mathbf{v} \in H$,the set of matrices whose elements are

$$A^{\mathbf{uv}}_{ij} = \overline{a_{ij}} \text{ if } u_i.v_j = -1, \underline{a_{ij}} \text{ if } u_i.v_j = 1$$

The set of matrices $\mathbf{A^{uv}}$ has 2^{2n-1} elements. If all matrices in the set have a determinant of the same sign, then all the matrices `A` are regular [22].

Note than another simplification procedure may be built by using the spectral radius (see section 7.6).

7.4.2.2 Implementation

Rohn test is implemented as:

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

the procedure returning -1 if all the matrices in `A` are regular, 0 otherwise.

To speed up the procedure a remembering mechanism has been implemented. If the flag `ALIAS_Rohn_Remembering` is set to 1 the procedure will store in the array `ALIAS_Rohn_Matrix` a set of at most `ALIAS_Rohn_Remembering_Step` matrices for which the Rohn test has already been done, together with at which step the test has failed (the calculation of the determinant of the scalar matrices is always done in the same order). This may allow to avoid a large number of determinant calculation.

7.4.3 Regularity of matrix with linear elements

7.4.3.1 Mathematical background

Let assume that a parametric matrix has rows (or columns) in which some elements depend linearly on some of the unknowns. Let consider the set H of interval matrices obtained as follows:

- for row i let A_i be the elements that depend linearly on the unknowns $\{x_k, x_l, \ldots\}$ (called the linear unknowns)
- construct all possible rows for row i by fixing the linear unknowns either to their minimum or maximum
- construct the set H by taking all possible combinations of all rows

It can be then shown that if all the matrices in H have their determinants of the same sign, then the set of parametric matrices does not include any singular matrix. This test is more powerful than the test proposed by Rohn (see section 7.4.2) as it takes more into account the structure of the parametric matrix.

7.4.3.2 Implementation

A regularity test based on this approach is implemented as:

```
int ALIAS_Check_Regularity_Linear_Matrix(int DimA,
   INTERVAL_VECTOR (* Func)(int l1, int l2,INTERVAL_VECTOR & v_IS),
   int (* A_Cond)(int dimA,
      INTERVAL_VECTOR (* Func1)(int l1, int l2,INTERVAL_VECTOR &v_IS),
      INTERVAL_VECTOR & v_IS,INTERVAL_MATRIX &A),
   int Row_Or_Column,
   int Context,
   INTEGER_MATRIX &Implication_Var,
   int Use_Rohn,
   INTERVAL_VECTOR &Domain)
```

where

- `Row_Or_Column`: 1 if the row of the matrix are used, 2 if the columns are used
- `Context`: is used to determine if this procedure is used according to the following rules (see section 7.4 for the meaning of the flag `Simp_in_Cond`):
  - always used if 100 or if `Context` is equal to `Simp_in_Cond`
  - not used if `Context` lie in [-2,2]
  - not used if `Context`=3 and `Simp_in_Cond` < 0
  - not used if `Context`=4 and `Simp_in_Cond` < 1
  - not used if `Context`=5 and `Simp_in_Cond` is not 0 or 1
  - not used if `Context`=6 and `Simp_in_Cond` is not 0 or 2
- `Implication_Var`: an integer matrix of dimension `DimA` $\times\ n$, where n is the number of unknowns. If this matrix has a 1 at row i, column j, then the unknown x_j appear linearly in some elements of row i (or column i) of the matrix A
- `Use_Rohn`: 1 if the Rohn consistency test is used to check that a matrix in H has a constant sign
- `Domain`: the ranges for the input parameters

This procedure return -1 if all elements of A are regular, 0 otherwise

7.5 Characteristic polynomial

It is possible to calculate the coefficients of the characteristic polynomial of a real or interval matrix using the procedures:

```
INTERVAL_VECTOR Poly_Characteristic(INTERVAL_MATRIX &A)
INTERVAL_VECTOR Poly_Characteristic(MATRIX &A)
```

In both cases the coefficients are returned as an interval vector which contains the coefficients of the polynomial by increasing order.

7.6 Spectral radius

Safe calculation of the spectral radius of a square real or interval matrix may be obtained with the procedures

```
int Spectral_Radius(INTERVAL_MATRIX &AA,double eps,double *ro,int iter)
int Spectral_Radius(MATRIX &AA,double eps,double *ro,int inter)
```

where

- `AA`: the matrix
- `eps`: a real that will be used to increment the solutions found for the median polynomial (i.e. the polynomial whose coefficients are the mid-point of the interval coefficients of the characteristic polynomial) until the polynomial evaluation does not contain 0
- `ro`: the upper bound of the spectral radius
- `iter`: the maximal number of allowed iteration

These procedures return 1 on success, 0 on failure (in which case increasing `eps` may be a good option) and -1 if the matrix is not square. The procedures

```
int Spectral_Radius(INTERVAL_MATRIX &AA,double eps,double *ro)
int Spectral_Radius(MATRIX &AA,double eps,double *ro)
```

may also be used with a maximum number of iteration fixed to 100.

If it intended just to show that the spectral radius is larger than a given value `seuil` then you may use

```
int Spectral_Radius(INTERVAL_MATRIX &A,double eps,double *ro,double seuil);
int Spectral_Radius(MATRIX &A,double eps,double *ro,double seuil);
int Spectral_Radius(INTERVAL_MATRIX &A,double eps,double *ro,int iter,double seuil);
int Spectral_Radius(MATRIX &A,double eps,double *ro,int iter,double seuil);
```

Note that the calculation of the spectral radius may be used to check the regularity of an interval matrix. Indeed let $\mathbf{A}$ be an interval matrix of dimension n, I_n the identity matrix of dimension n. If $\mathbf{A_c}$ is the mid-matrix of $\mathbf{A}$ we may write

$$\mathbf{A} = [\mathbf{A_c} - \Delta, \mathbf{A_c} + \Delta]$$

Let $\mathbf{R}$ be an arbitrary matrix. It can be shown that if

$$\rho(|I_n - \mathbf{R}\mathbf{A_c}| + |\mathbf{R}|\Delta) < 1$$

where ρ denotes the spectral radius, then $\mathbf{A}$ is regular [22]: this is known as the Beeck-Ris test.

Chapter 8

Optimization

ALIAS is also able to deal with optimization problem. Let consider a scalar function F of m variables $X = \{x_1, \ldots, x_m\}$, called the *optimum function* and assume that you are looking for the global minimal or maximal value of F when the unknowns lie in some ranges. Furthermore the unknowns may be submitted to n_1 constraints of type $G_i(X) = 0$, n_2 constraints of type $H_i(X) \leq 0$ and n_3 constraints of type $K_i(X) \geq 0$. Note that F, G_i, H_i, K_i may have interval coefficients.

8.1 Definition of a minimum and a maximum

As interval coefficients may appear in the function F we have to define what will be called a minimum or a maximum of F. First we assume that there is no interval coefficients in F and denote by F^m the minimal or maximal value of F over a set of ranges defined for X and an accuracy ϵ with which we want to determine the extremum. The algorithm will return an interval F_e as an approximation of F^m such that for a minimization problem $\underline{F_e} - \epsilon \leq F^m \leq \underline{F_e} + \epsilon$ and for a maximization problem $\overline{F_e} - \epsilon \leq F^m \leq \overline{F_e} + \epsilon$. The algorithm will also return a value X^m for X where the extremum occurs. If we deal with a constrained optimization problem we will have:

- $-\epsilon_f \leq G_i(X^m) \leq \epsilon_f$
- $H_i(X^m) \leq 0$ or $0 \in H_i(X^m)$ and $\overline{H_i(X^m)} - \underline{H_i(X^m)} \leq \epsilon_f$
- $K_i(X^m) \geq 0$ $0 \in K_i(X^m)$ and $\overline{K_i(X^m)} - \underline{K_i(X^m)} \leq \epsilon_f$

where ϵ_f has a pre-defined value. Note that if we have constraint equation of type $G_i(X) = 0$ the result of the optimization problem may be no more guaranteed as the constraint itself G_i may not be verified.

If there are interval coefficients in the optimum function there is not a unique F^m but according to the value of the coefficient a minimal extremum value F_1^m and a maximal extremum value F_2^m. The algorithm will return in the lower bound of F_e an approximation of F_1^m and in the upper bound of F_e an approximation of F_2^m which verify for a minimization problem:

- $\underline{F_e} - \epsilon \leq F_1^m \leq \underline{F_e} + \epsilon$
- $\overline{F_e} - \epsilon \leq F_2^m \leq \overline{F_e} + \epsilon$

Note that the width of F_e may now be greater than ϵ.The algorithm will return also two solutions X_1^m, X_2^m for X corresponding respectively to the values of F_1^m, F_2^m. If we are dealing with a constrained optimization problem the solutions will verify the above constraint equations.

If the optimum function has no interval coefficients the algorithm may return no solution if the interval evaluation of the optimum function has a width larger than ϵ. Evidently the algorithm will also return no solution if there is no solution that satisfy all the constraints.

8.2 Methods

A first method to find *all* the solutions of an optimum problem is to consider the system of derivative equations and find its root (eventually under the given constraints): this may be done with the solving procedures described in a previous chapter. Hence we have implemented an alternative method which is able to work even if the optimum function has, globally or locally, no derivatives. This method is implemented as a special case of the general solving procedures. The only difference is that the procedure maintains a value for the current optimum: during the bisection process we evaluate the optimum function for each new box and reject those that are not compatible with the current optimum.

Two types of method enable to solve an optimization problem:

1. a method which need only a procedure for evaluating F,
2. a method which need a procedure for evaluating F and a procedure for evaluating its gradient,

Note that with the first method none of the function needs to be differentiable while for the other one not all the functions in the set F, G, H, K must be differentiable: only one function in the set has to be differentiable.

For the first method we update the current optimum by computing the optimum function value at the middle point of the box. For the method with the gradient a local optimizer based on the steepest descent method is also used if there is only one equation to be minimized or if there are only inequalities constraints. The local optimizer works in 2 steps: first a rough optimization and then a more elaborate procedure if the result of the first step is better than a fixed threshold defined by the global variable `ALIAS_Threshold_Optimiser` whose default value is 100. Additionally if an extremum E has been already determined the local optimizer (which may computer intensive) is called only if the function value V at the middle point is such that for a maximum $|V| \geq$ `ALIAS_LO_A`$|S|$ and $|V| \leq$ `ALIAS_LO_B`$|S|$ for a minimum, where `ALIAS_LO_A` and `ALIAS_LO_B` are global variables with default value 0.9 and 1.1.

8.3 Implementation

Preliminary notes:

- for both procedures you may set the global variable `ALIAS_Has_Optimum` to 1 to indicate that you have already determined a possible optimum value. This value has to be given in the global interval variable `ALIAS_Optimum`. Note that this variable is used to store the minimum or the maximum of the function to be minimized or maximized. If no a-priori information on the extremum has been given `ALIAS_Has_Optimum` will be set to 1 as soon as the algorithm has a current estimation of the extremum. If both the minimum and maximum are looked for, then this variable will be used to store the minimum while `ALIAS_Optimum_Max` will be used to store the maximum. If you have indicated an a-priori estimation of the extremum it may happen that the algorithm is unable to find a better extremum. In that case the global variables `ALIAS_Algo_Optimum`, `ALIAS_Algo_Optimum_Max` will be set to the extremum find by the algorithm (hence if no a-priori information has been given `ALIAS_Algo_Optimum` and `ALIAS_Optimum` will be the same). The flag `ALIAS_Algo_Has_Optimum` will be set to 1 as soon as the algorithm has a current estimation of the extremum (which may be worse than the optimum given a priori). If the algorithm find a better optimum than the optimum given a priori the flag `ALIAS_Algo_Find_Optimum` will be set to 1.

  The location of the minimum and maximum can be found in the interval matrix `ALIAS_Vector_Optimum`, the first line indicating the location of the minimum.

- the flag `ALIAS_Optimize` is used to indicate which type of optimum you are looking for: -1 for a minimum, 1 for a maximum and 10 for both. This flag is automatically set by the optimization procedures

- the accuracy with which the optimum will be computed is available in the global variable `ALIAS_Accuracy_Extremum`

- the function number of the expression to be optimized is available in the global variable `ALIAS_Opt_Func`

- you may follow the progress of the optimization process by setting the debug flag to at least 1 and setting the flag `ALIAS_Allow_Storage` to 1. In that case each time the algorithm find a better approximation than the current one he will write down the new value in the file `ALIAS_Allow_Storage_File` (whose default name is `.opti`)
- you may also use these procedures just to test if a given function has a minimum lower or a maximum greater than a given value. If you set the flag `ALIAS_Stop` to 1 and the double `ALIAS_Extremum` to the desired value the optimization procedure will return as soon as a better minimum or maximum is found. Note that the optimization procedure always reset `ALIAS_Stop` to 0.
- these procedures are only a specific cases of the general solving procedures: hence the computation time may be largely influenced by the bisection mode
- another factor which may be of importance for the efficiency is the storage mode. We have seen that two storage modes are available in `ALIAS`: the direct or reverse modes. The underlying principle in the interval analysis method is to examine the leaves of a tree, the terminal leaves satisfying a decision criteria: either they are a solution of the problem or not. In the reverse storage mode we follow all the branches starting from a certain leave until we reach all the terminal leaves of obtained from this leave while in the direct storage mode we skip from one branch to another. This mode has the drawback of an exponential growth in the needed storage while the reverse mode has the advantage to go directly to the smallest leaves that may be deleted, thereby freeing storage space. When solving a system the tree is *static* i.e. we can construct it (this is what the algorithm is doing) and we explore all the leaves. In an optimization problem the tree is *dynamic*: as soon as the exploration of a leave enable to update the current estimation of the extremum we in fact all the branches of the tree that cannot lead to an improvement of the extremum. In other words it is important to determine as quickly as possible the best estimation. In an optimization problem we cannot determine beforehand the branches that has to be followed for finding the leaves which is the solution. Thereby the reverse mode has the drawback that if we are following the wrong branches we will never eliminate branches and we will explore a large number of leaves that may have been avoided. With the direct mode we skip from one branch to another and thus we have a good probability of updating quickly the extremum. It will therefore be of good policy to use the direct mode but we then have to deal with the problem of storage. A mixed strategy can be used: we start with the direct mode and as soon as we expect a storage problem we switch to the reverse mode. This may be done by using the flag `Switch_Reverse_Storage` to a value s larger than 1: in that case as soon as the number of stored unexplored boxes exceed s the algorithm will switch to the reverse mode. The value of s may, for example, be chosen as half the number of storage space that is allocated. In this optimization mode we may also use a mixed strategy between the direct and reverse storage mode: if the flag `Reverse_Storage` is set to $n \geq 1$, then if the bisection of the current box leads to m new boxes, the n first boxes (as ordered using the ordering described in the **Order** section) are stored in the list starting at the current position of the list, while the $m - n$ remaining boxes are stored at the end of the list. Thus setting n to 1 is equivalent to using the direct storage mode, while, if we have k unknowns, setting n to a value greater than k is equivalent to using the reverse storage mode.
- if you use a mixed bisection mode there is a convenient way to indicate some dependencies between the variable, see the note 8.3.5
- the 2B method (see section 2.17) may be used to improve the efficiency of the optimization algorithms. The ALIAS-Maple procedures `HullIConsistency`, `HullConsistency` implement the 2B method and allows to deal with an equation or an inequality that involves the current optimum.

8.3.1 Optimization with function evaluation

The optimization method is implemented as:

```
int Minimize_Maximize(int m,int n,
      INTEGER_VECTOR &Type_Eq,
      INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
      INTERVAL_VECTOR & TheDomain,
```

```
int Iteration,int Order,
double epsilon,double epsilonf,double epsilone,
int Func_Has_Interval,
INTERVAL Optimum,
INTERVAL_MATRIX & Solution,
int (* Simp_Proc)(INTERVAL_VECTOR &));
```

the arguments being:

- `m`: number of unknowns
- `n`: number of equations, see the note 2.3.4.1
- `Type_Eq`: type of the equations:
  - `Type_Eq(i)`=-1 if equation `i` is a constraint equation of type $H(X) \leq 0$
  - `Type_Eq(i)`=0 if equation `i` is a constraint equation of type $G(X) = 0$
  - `Type_Eq(i)`=1 if equation `i` is a constraint equation of type $K(X) \geq 0$
  - `Type_Eq(i)`=-2 if equation `i` is the optimum function to be minimized
  - `Type_Eq(i)`=2 if equation `i` is the optimum function to be maximized
  - `Type_Eq(i)`=10 if equation `i` is the optimum function for which is sought the minimum and maximum
- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see the note 2.3.4.3. This function must be written in a similar manner than for the general solving procedures. Note also that a convenient way to write the `IntervalFunction` procedure is to use the possibilities offered by the ALIAS-Maple (see the ALIAS-Maple manual).
- `TheDomain`: box in which we are looking for the extremum of the optimum function
- `Iteration`: the number of boxes that may be stored
- `Order`: a flag describing which order is used to store the new boxes, see the note 8.3.4
- `epsilon`: the maximal width of the solution intervals but not used. Should be set to 0.
- `epsilonf`: the maximal error for the equality constraints. If the problem has constraint of type $G(X) = 0$ then a solution will verify $-\texttt{epsilonf} \leq G(X) \leq \texttt{epsilonf}$
- `epsilone`: the maximal error on the extremum value. If the extremum of the function is G_1 and the procedure returns the value G, then a minimum will verify $G - \texttt{epsilone} \geq G_1$ and a maximum $G_1 \leq G + \texttt{epsilone}$.
- `Func_Has_Interval`: 1 if the optimum function has interval coefficients, 0 otherwise
- `Optimum`: an interval which contain the extremum value of the optimum function
- `Solution`: an interval matrix of size at least (2,`m`) which will contained the values of X for which the extremum are obtained
- `Simp_Proc`: an optional parameter which is a simplification procedure that may be provided by the user. It takes as input a box P and may:
  - either returns in P a box with lower width than the initial P and a return code 0 or 1
  - or indicates that there is no solution to the optimization problem in the current box, in which case the procedure returns -1

  An often efficient simplification procedure is the 2B method (see section 2.17) that may be automatically generated by the `HullIConsistency` procedure of ALIAS-Maple

Thus to minimize a function you have to set its Type_Eq to -2, to maximize it to set its Type_Eq to 2, while if you are looking for both a minimum and a maximum Type_Eq should be set to 10.

Remember that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2) if you have constraint equations.

In some cases it may be interesting to determine if the minimum and maximum have same sign. This may be done by setting the flag ALIAS_Stop_Sign_Extremum either to:

- 1: the procedure will return immediately as soon as it is proven that the extremum will have opposite sign i.e. as soon as two points lead to opposite value for the function. But if the extremum have identical sign the minimum and maximum will be computed up to the accuracy epsilone
- 2: the procedure will return immediately as soon as it is proven that the extremum will have opposite sign i.e. as soon as two points lead to opposite value for the function. If the extremum have same sign the values returned by the procedure are not the minimum and maximum of the function

With the flag set to 2 the detection that the extremum will have opposite sign is faster.

8.3.1.1 Return code

The procedure will return an integer k that will be identical to the code returned by the procedure Solve_General_Interval except for:

- $k = -4$: one of the equation in the system has not a type 0, -1, -2, 2, 10 or 1 (i.e. it's not an equation or an optimum function, neither inequality $F \leq 0$ or an inequality $F \geq 0$)
- $k = -10$: there is no optimum function i.e. no equation has type 2 or -2 or 10
- $k = -20$: there is more than one optimum function i.e. more than one equation has type 2 or -2 or 10

8.3.1.2 Dealing with inequalities on the same function

In this version of ALIAS there is no direct way to deal with inequalities that are valid for the same function (e.g. $\alpha \leq U(X) \leq \beta$), which mean that you will have to declare two inequalities (which imply that the quantity U will be evaluated twice). But in the previous procedure there is a way to avoid writing two inequalities. In the function evaluation procedure you will just compute $U(X)$ and declare this function as an inequality of type $H(X) \leq 0$. After having computed the interval evaluation U

- if this interval is strictly included in $[\alpha, \beta]$ substitute the interval of U by the value -1 (or any negative number)
- if this interval has no intersection with $[\alpha, \beta]$ substitute the interval of U by the value 1 (or any positive number)
- if this interval has an intersection with $[\alpha, \beta]$ but is not strictly included in it, then substitute the interval of U by the interval [-1,1]

8.3.2 Optimization with function and jacobian evaluation

The optimization method is implemented as:

```
int Minimize_Maximize_Gradient(int m,int n,
      INTEGER_VECTOR &Type_Eq,
      INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
      INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
      INTERVAL_VECTOR & TheDomain,
      int Iteration,int Order,
      double epsilon,double epsilonf,double epsilone,
      int Func_Has_Interval,
      INTERVAL Optimum,
```

```
    INTERVAL_MATRIX & Solution,
    int (* Simp_Proc)(INTERVAL_VECTOR &));
```

the arguments being:

- `m`: number of unknowns
- `n`: number of equations, see the note 2.3.4.1
- `Type_Eq`: type of the equations:
  - `Type_Eq(i)`=-1 if equation `i` is a constraint equation of type $H(X) \leq 0$
  - `Type_Eq(i)`=0 if equation `i` is a constraint equation of type $G(X) = 0$
  - `Type_Eq(i)`=1 if equation `i` is a constraint equation of type $K(X) \geq 0$
  - `Type_Eq(i)`=-2 if equation `i` is the optimum function to be minimized
  - `Type_Eq(i)`=2 if equation `i` is the optimum function to be maximized
  - `Type_Eq(i)`=10 if equation `i` is the optimum function for which is sought the minimum and maximum
- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see the note 2.3.4.3. This function must be written in a similar manner than for the general solving procedures.
- `Gradient`: a function which return the interval evaluation of the gradient of the equations, see the note 2.4.2.2. This function must be written in a similar manner than for the general solving procedures **with the additional constraint** that the function to be minimized of maximized must be the last one.
- `TheDomain`: box in which we are looking for the extremum of the optimum function
- `Iteration`: the number of boxes that may be stored
- `Order`: a flag describing which order is used to store the new boxes, see the note 8.3.4
- `epsilon`: the maximal width of the solution intervals but not used. Should be set to 0.
- `epsilonf`: the maximal error for the equality constraints. If the problem has constraint of type $G(X) = 0$ then a solution will verify $-$`epsilonf` $\leq G(X) \leq$ `epsilonf`
- `epsilone`: the maximal error on the extremum value. If the extremum of the function is G_1 and the procedure returns the value G, then a minimum will verify $G -$ `epsilone` $\geq G_1$ and a maximum $G_1 \leq G +$ `epsilone`.
- `Func_Has_Interval`: 1 if the optimum function has interval coefficients, 0 otherwise
- `Optimum`: an interval which contain the extremum value of the optimum function
- `Solution`: an interval matrix of size at least (2,`m`) which will contained the values of X for which the extremum are obtained
- `Simp_Proc`: an optional parameter which is a simplification procedure that may be provided by the user. It takes as input a box P and may:
  - either returns in P a box with lower width than the initial P and a return code 0 or 1
  - or indicates that there is no solution to the optimization problem in the current box, in which case the procedure returns -1

Thus to minimize a function you have to set its `Type_Eq` to -2 and to maximize it to set its `Type_Eq` to 2.

Remember that you may use the 3B method to improve the efficiency of this algorithm (see section 2.3.2).

Note also that a convenient way to write the `IntervalFunction` and `Gradient` procedures is to use the possibilities offered by ALIAS-Maple (see the ALIAS-Maple manual).

8.3.3 Return code

The procedure will return an integer k that will be identical to the code returned by the procedure `Solve_General_Gradient_Interval` except for:

- $k = -4$: one of the equation in the system has not a type 0, -1, -2, 2, 10 or 1 (i.e. it's not an equation or an optimum function, neither inequality $F \leq 0$ or an inequality $F \geq 0$)
- $k = -10$: there is no optimum function i.e. no equation has type 2 or -2 or 10
- $k = -20$: there is more than one optimum function i.e. more than one equation has type 2 or -2 or 10
- $k = -30$: the last function is not the one to be minimized or maximized

Note that this implementation is only a special occurrence of the general solving procedure and thus offer the same possibilities to improve the storage management (see section 2.4).

8.3.4 Order

8.3.4.1 General principle

During the bisection process new boxes will be created and stored in the list. But we want to order these new boxes so that the procedure will consider first the most promising box. The ordering is based on an evaluation index, the new boxes being stored using an increasing order of the index (the box with the lowest index will be stored first). The flag `Order` indicate which index is used: indexMAXCONSTRAINTFUNCTION@`MAX_CONSTRAINT_FUNCTION`

- `MAX_FUNCTION_ORDER`: let $[\underline{F_i}, \overline{F_i}]$ be the interval evaluation of the optimum equation i. The index is obtained as $\underline{F_i}$ for a minimization problem and $-|\overline{F_i})|$ for a maximization problem,
- `MAX_CONSTRAINT_FUNCTION`: same than `MAX_FUNCTION_ORDER` if there is only one equation in the system. Otherwise:
  - if there are equality constraints use the `MAX_FUNCTION_ORDER` index
  - if there are only inequality constraints and if they all hold for all the new boxes, then the index is the lower bound of the optimum function for a minimization problem and the upper bound for a maximization problem.
  - if there are only inequality constraints and none of the new boxes satisfied them all: the index is the upper bound of the inequality for the constraint of type ≤ 0 and the absolute value of the lower bound for the constraint of type ≥ 0
  - if there are only inequality constraints and they are all verified only for some of the new boxes, then the index will be calculated in such way that the boxes satisfying the constraints will be stored first according to the value of the lower or upper bound of the optimum function. Then will be stored the boxes not satisfying the constraints according to the index described in the previous item
- `MAX_MIDDLE_FUNCTION_ORDER`: let F_i^m be the value of the function F_i computed for the middle point of the box. The index is the absolute value of F_i^m

8.3.5 The variable table

Assume now that you have chosen a mixed bisection in which the bisection is applied on $n > 1$ variables. The procedure will choose the bisected variables using, for example, the smear function. But in some cases it may be interesting to guide the bisection: for example if we know that subsets of the variables have a strong influence on the extremal value of the optimum function it may be interesting to indicate that as soon as the smear function has led to bisecting one variable in a given subset it may be good to bisect also the other variables in the subset. For example consider the following functions:

$$f_i = u_x + a\cos\theta_i + b\cos(\theta_i + \psi_i) - x_i$$
$$g_i = u_y + a\sin\theta_i + b\sin(\theta_i + \psi_i) - y_i$$

where $u_x, u_y, a, b, \theta_i, \psi_i$ are unknowns and x_i, y_i are given. Consider now the optimum function F:

$$F = \sum_{i=1}^{i=10} f_i^2 + g_i^2$$

which has 24 unknowns. But clearly each subset (θ_i, ψ_i) has a strong influence on the minimum of F. Hence if one of the θ_i is bisected it may be interesting to bisect also ψ_i. This may be done by setting the flag `ALIAS_Guide_Bisection` to 1 and using the *variables table*: for a problem with m unknowns the variables table V is an array of size $m \times m$ and a 1 in $V(i, j)$ indicates that if the variable i is bisected then the variable j should be also bisected. In `ALIAS` the variables table is implemented under the name `ALIAS_Bisecting_Table`. It is the responsibility of the user to clear this array and update it as in the following example:

```
Resize(ALIAS_Bisecting_Table,24,24);
Clear(ALIAS_Bisecting_Table);
ALIAS_Bisecting_Table(1,2)=1;
ALIAS_Bisecting_Table(2,1)=1;
ALIAS_Bisecting_Table(3,4)=1;
ALIAS_Bisecting_Table(4,3)=1;
```

8.4 Examples

8.4.1 Example 1

Consider the equation: $\cos(x) + y\cos(y) - 0.2 = 0$ which defines a curve in the $x - y$ plane on which you want to determine the closest point to the origin when x, y lie in the range $[-\pi, \pi]$. This leads to trying to find the minimum of the function $x^2 + y^2$ under the constraint $\cos(x) + y\cos(y) - 0.2 = 0$.

The procedure for the interval evaluation of the 2 functions will be written as:

```
INTERVAL_VECTOR IntervalTestFunction (int l1,int l2,INTERVAL_VECTOR & in)
// interval valued test function
{
INTERVAL x,y;
INTERVAL_VECTOR xx(2);

x=in(1);
y=in(2);
if(l1==1)
  xx(1)=Cos(x)+y*Sqr(Cos(y))-0.2;
if(l1<=2 && l2>=2)
  xx(2)=Sqr(x)+Sqr(y);
  return xx;
}
```

while the main program may be written as:

```
int main()
{
int Iterations, Dimension,Dimension_Eq,Num,i,j,precision;
double Accuracy,Accuracy_Variable;
INTERVAL_MATRIX SolutionList(2,2);
INTERVAL_VECTOR TestDomain(2),F(2),P(2),H(2);
INTEGER_VECTOR Type(2);
INTERVAL Optimum;
REAL pi;

pi=Constant::Pi;

Dimension_Eq=2;Dimension=2;
TestDomain(1)=INTERVAL(-pi,pi);TestDomain(2)=INTERVAL(-pi,pi);

cerr << "Number of iteration = "; cin >> Iterations;
cerr << "Accuracy on Function = "; cin >> Accuracy;

Type(1)=0;Type(2)=-2;
Accuracy=0;

Num=Minimize_Maximize(Dimension,Dimension_Eq,Type,
       IntervalTestFunction,TestDomain,Iterations,Accuracy_Variable,
       Accuracy,0,Optimum,SolutionList);
if(Num<0)
  {
    cout << "The procedure has failed, error code:"<<Num<<endl;
    return 0;
```

```
  }
cout<<"Optimum:"<<Optimum<<" obtained at"<<endl;
for(i=1;i<=Num;i++)
  {
    cout << "x=" << SolutionList(i,1) <<endl;
    cout << "y=" << SolutionList(i,2) <<endl;
  }
  return 0;
}
```

The `Minimize_Maximize` and `Minimize_Maximize_Gradient` procedures will return the same numerical results but the number of boxes will change. The results obtained for a full bisection,
the `MAX_MIDDLE_FUNCTION_ORDER` and according to the accuracy `epsilonf` and the storage mode (either direct (DSM) or reverse (RSM), see section 2.3.1.2) are presented in the following table (the number of boxes for the `Minimize_Maximize_Gradient` procedure is indicated in parenthesis):

`epsilonf`	Minimum	X_1^m	X_2^m	boxes
0.01, DSM	[1.12195667, 1.12195667]	[-0.944932,-0.4786]	[-0.944932,-0.4786]	76 (36)
0.01, RSM	[1.12195667, 1.12195667]	[-0.944932,-0.4786]	[-0.944932,-0.4786]	59 (37)
0.001, DSM	[1.1401661,1.1401661]	[-0.954903,-0.477835]	[-0.954903,-0.477835]	201 (75)
0.001, RSM	[1.1401661,1.1401661]	[-0.954903,-0.477835]	[-0.954903,-0.477835]	148 (67)
0.000001, DSM	[1.14223267,1.14223267]	[-0.957596,-0.474596]	[-0.957596,-0.474596]	5031 (2041)
0.000001, RSM	[1.14223267,1.14223267]	[-0.957596,-0.474596]	[-0.957596,-0.474596]	4590 (2164)

This example shows clearly the influence of the constraint equation on the determination of the optimum.

8.4.2 Example 2

Consider the problem of finding the coordinates x, y, z of a point that lie on the surface $x^3 + y^3 + z^3 + 1 - (x + y + z + 1)^2$, is inside the sphere centered at (-1,-1,0) of radius 1 and is the closest possible to the center of this sphere, with the constraint that x, y, z lie in the range [-2,2]. Thus we have:

$$F(X) = (x+1)^2 + (y+1)^2 + z^2$$
$$G(X) = x^3 + y^3 + z^3 + 1 - (x+y+z+1)^2 = 0$$
$$H(X) = (x+1)^2 + (y+1)^2 + z^2 - 1 \leq 0$$

With `epsilonf`=0.0001 we find out that the point is located at at (-0.747,-0.747,0.086059) which is well inside the sphere and that the minimal distance is 0.13529.

We may also compute the minimal distance not to a point but to a line segment, for example defined by $x \in [0.9, 1.1]$, $y = -1$, $z = 0$. In that case the optimum function in the evaluation procedure may be defined as:

```
Sqr(x+INTERVAL(0.9,1.1))+Sqr(y+1)+Sqr(z)
```

and with `epsilonf`=0.0001 the algorithm will return that the minimal distance lie in the range [0.0907,0.1925].

Chapter 9

Continuation for one dimensional system

Although mostly devoted to zero-dimensional systems ALIAS still offers some possibilities to deal with n-dimensional systems. The purpose of the algorithms here is to consider a system with n parameter and to follow the branches of the solutions of the system when these parameters are varying between some bounds.

9.1 Continuation 1D

9.1.1 Mathematical background

Let consider a system of n equations in m unknowns $F_i(x_1, \ldots, x_m) = 0$ and assume that x_1 may vary in the range $[\underline{x_1}, \overline{x_1}]$ and that it is considered as the parameter of this system. Using the solving algorithms of ALIAS (or any other method) we are able to determine the real roots of this system, if any, for a given value of the parameter, for example for $x_1 = \underline{x_1}$. Let assume that we have found k real roots $X_1, \ldots, X_k$. Assume now that we change the value of x_1 to $\underline{x_1} + \epsilon$. Using Kantorovitch theorem (see section 3.1.2) we are able to determine if Newton method with as initial estimate X_i will converge toward a solution of the new system and the radius of convergence α_i such that the solution is unique in the ball $B_i = [X_i - \alpha_i, X_i + \alpha_i]$ (alternatively we may also use Moore theorem, see section 2.10). If this is the case and if the intersection of the B_i is empty, then we are able to compute in a certified way the new k solutions of the new system. Otherwise we will repeat the procedure with a smaller value for ϵ until the method succeed. A failure case will be when the system become singular at a point (for example when 2 branches are collapsing). If this happen we will change the value of x_1 until we find a new set of solutions which satisfy Kantorovitch theorem and start again the process.

This method enable to follow all the branches for which initial points have been found when solving the initial system. We may then store these points in an array, for example at regular step for the value of the parameter.

By default we use Kantorovitch theorem to follow the branches but by setting the global variable `ALIAS_Use_Kraw_Continuation` to 1 we will use Moore test.

9.1.2 Implementation

9.1.2.1 Certified Newton

The procedure that allow to determine the next point on a branch is implemented as:

```
int Certified_Newton(int Nb_Var,int Dimension_Eq,int Nb_Branch,int Branch,
                       INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                       INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
                       INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),
                       double Accuracy_Var,double Accuracy,
                       double *param,double delta_param,double min_delta_param,
```

```
                              int sens,MATRIX &Solution)
```

This procedure determine if it possible to find a new point on the branch numbered `Branch` among the `Nb_Branch` that have been found. In `Solution` we have all the solutions that have been found for the branches when the parameter value was `param`. The change in the parameter value will be at most `sens`×`delta_param` and on exit of the procedure `param` will be set to the new value. `sens` indicates the direction of change in the parameter value: +1 for an increase, -1 for a decrease.

This procedure returns:

- -1, -4: even with a change of the parameter value of `min_delta_param` we cannot either distinguish two solutions or Kantorovitch has failed to give a positive answer. But this does not mean in general that a singularity occurs.
- -2: for systems having more equations than unknowns the solution that has been obtained with Newton for the square system of equations failed to cancel the remaining equations
- -3: Newton has not converged (should not occur)
- -10: singular point
- 1, 2: procedure has succeeded

9.1.2.2 Procedure for following branches

This procedure takes as input a set of n solutions of the system and will return points on the n branches. The branches will be followed until a given value for the parameter is reached or if Kantorovitch theorem is no more satisfied for some value of the parameter. It is implemented as:

```
int ALIAS_Start_Continuation(int m,int n,int NUM,
       INTERVAL_MATRIX &Solutions,
       INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* IntervalHessian)(int,int,INTERVAL_VECTOR &),
       double epsilon,double epsilonf,
       double *z,double delta,double mindelta,
       INTERVAL &Rangez,
       int sens,
       MATRIX &BRANCH,int *NBBRANCH)
```

the arguments being:

- `m`: number of unknowns
- `n`: number of equations
- `NUM`: the number of solutions of the system for the current value of the parameter `z`
- `Solutions`: the solutions of the system for the current value of the parameter `z`
- `IntervalFunction`: a function which return the interval vector evaluation of the equations, see note 2.3.4.3.
- `IntervalGradient`: a function which return the interval matrix of the jacobian of the equations, see note 2.4.2.2
- `IntervalHessian`: a function which return the interval matrix of the Hessian of the equations, see note 2.5.2.1
- `epsilon`: the maximal width of the box, see the note 2.3.4.6. If all the variable ranges have a width lower than this value and the interval evaluation of the equations contains all 0, then the set of ranges is considered to be a solution. But they will be not considered as a valid solution as they will not satisfy Kantorovitch theorem. Hence you must put here a very small value

- `epsilonf`: the maximal width of the equation intervals, see the note 2.3.4.6. This value will be used by the iterative Newton scheme to stop the iteration.
- `z`: the parameter of the system
- `delta`: if z_0 is the first value of the parameter such that the system has solutions, then we will store in `BRANCH` the solutions for z_0+`delta`, z_0+2`delta`,...
- `mindelta`: if the algorithm is unable to find a Kantorovitch solution for the parameter value `z+mindelta` while a solution has been found for `z`, then the algorithm will assume that we are close to a singular point. Hence `mindelta` should have a small value
- `Rangez`: the range for the parameter `z`
- `sens`: either 1 or -1. If 1 the branch will be computed for increasing values of `z`, if -1 for decreasing values of `z`. This parameter should be chosen according to the largest number of solutions of the system for the lower and upper value of `z`
- `BRANCH`: the procedure will return an array of `NBBRANCH` lines and `m+2` columns, which describe the points of the branch. In a line the first `m`+1 elements are the coordinates of the point and the `m`+2 elements is the number of the branch to which belong this point. The points are not ordered with respect to the branch number. The algorithm will take care of resizing `BRANCH` as needed, hence there is no need to give dimension for this parameter
- `NBBRANCH`: the total number of points in the array `BRANCH`.

The return code is:

- 1: the branches have been successfully determined
- -1, -2: the algorithm has failed to find a Kantorovitch solution for the current value of the parameter
- -3: failure in Newton method (should not occur)
- -10: for the current value of the parameter the jacobian is singular
- -20: `sens` is not 1 or -1
- -30: `delta` or `mindelta` is negative

Even for a negative return code the points in `BRANCH` are valid: the procedure will return the points that have computed on the n branches but is simply not able to determine the location of new branches that may have appeared.

9.1.2.3 Full continuation procedure

This procedure will determine initial points on the branches of the system for the value of the parameter within some bounds and then follow the branches. It is implemented as

```
int ALIAS_Full_Continuation(int m,int n,
       INTERVAL_VECTOR (* IntervalFunction)(int,int,INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
       INTERVAL_MATRIX (* IntervalHessian)(int,int,INTERVAL_VECTOR &),
       INTERVAL_VECTOR &Domain,
       int M,
       double epsilon,double epsilonf,
       double *z,double delta,double mindelta,double mindz,
       INTERVAL &Rangez,
       int sens,
       MATRIX &BRANCH,int *NBBRANCH)
```

The arguments are the same than for the previous procedure except for:

- `Domain`: the range for the `m` variable in which we will look for solutions of the system,
- `M` is the maximum number of boxes which may be stored (see the note 2.3.4.5)
- `mindz`: the accuracy with which the starting point for the branches will be determined: if the value of the parameter at which the branches will start is z_0, then the system has no solution for $z = z_0$-`mindz`. This is done by using a bisection on the parameter: if at z_1 the system has no solution and has a positive number of solution at z_1+`delta`, then we will solve the system for z_1+`delta`/2. At each time we will store the value z_n of the parameter for which there is no solution of the system and the value z_y for which we have solutions and we will stop the bisection on the parameter as soon as $|z_n - z_y| <$ `mindz`.

To determine the starting points of the branches this procedure uses the solving procedure with the Hessian (see section 2.5). As soon as initial valid solutions are found the `ALIAS_Start_Continuation` procedure is called until either a bound of the parameter range is reached, or a singularity occur. In the later case the solving procedure is called with an increased value for the parameter. This algorithm cannot find isolated points and may miss branches (see next section).

There is also another version of this program where you indicate just before `Domain` the solutions which have already been found. The syntax is

```
INTERVAL_MATRIX (* IntervalHessian)(int,int,INTERVAL_VECTOR &),
int NUM,
INTERVAL_MATRIX &Solutions,
INTERVAL_VECTOR &Domain
```

The return code for these procedures are:

- $n > 0$: the number of branches found by the algorithm
- -1: no initial point have been found
- -10: Newton algorithm has failed (should not occur)
- -20: `sens` is not 1 or -1
- -30: `delta` or `mindelta` is negative

Finding the initial starting point with the accuracy `mindz` may be computer intensive. Hence the integer global variable `ALIAS_Allow_Backtrack` enable to disable this process if it is set to 0 (its default value is 1): in that case as soon as starting points have been found (hence at z_0+ n`delta`) we will start following the branches.

In fact these procedures are special occurrences of another `ALIAS` procedure which has another argument right after the `Hessian` argument. Assume for example that you are considering a system which has one equation written as:

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

where l_{25} is the parameter of the system and x, y the unknowns. When using the continuation method we have to define ranges for these unknowns in order to be able to solve the system of equations. Up to now we have indicated bounds that are constants but for the equation example it will be interesting to be able to specify that these bounds may change according to the value for the parameter using a *simplification procedure*. In our example clearly x and y cannot exceed $l_{25}+1$ and cannot be lower than $-l_{25}+1$ (this is an application of the concept of 2B-consistency, see section 2.17). Hence we may specify right after the `Hessian` argument the name of a procedure, for example `Range`, that is able to update the bounds on the unknowns according to the value of the parameter (or any other variable that may play a role). The syntax of the procedure `Range` is:

```
INTERVAL_VECTOR Range(double z, INTERVAL_VECTOR &Variable)
```

where `z` is parameter value and `Variable` the current set of ranges for the unknowns. This procedure must return a set of ranges for the unknowns (be careful to check that the returned ranges are included in the initial ranges).

Note also that the `ALIAS`-Maple package offers a procedure that uses the method described in section 2.14 for finding the initial starting points of the branches: this method is efficient if the equations include linear terms in the unknowns.

9.1.2.4 Missed branches

Clearly there is a major problem with the method we are proposing: we may miss some branches. For example imagine that a system has n roots for the initial value of the parameter, but will have more than n solutions for another value of the parameter, meaning that new branches appear: our algorithm will find only the n initial branches. There are two methods that can be used to find the correct number of branches. The first one is simply to start following the branches with as initial value for the parameter the one among the highest or smallest value having led to the maximum number of solution. There is also another mechanism that enable to avoid missing branches. Assume that the solving procedure has determined for some initial value λ_0 of the parameter n solutions to the system and that at some point λ_1 the continuation method has failed: the solving procedure is called and determine that the system has now $m > n$ solutions. This mean that for some parameter value between λ_0 and λ_1 we have missed $m - n$ branches. At such point, called *problem point* it would be interesting to backtrack i.e. to start again a continuation process with as initial point for the branches the m solutions obtained for λ_1 and a value for `Sens` which is the opposite of the initial value. This is what is done by the procedure which may store up to 10 problem points. As this process may be computer intensive it is possible to disable it by setting the integer global variable `ALIAS_Problem_Continuation` to -1 (it's default value is 0, which mean that the process is enabled).

9.1.3 Example

Let consider the system:

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

The purpose of this example is to write the points of the branches of this system in files whose name is `BRANCH` followed by the branch number. The main file is written as:

```
#include "Functions.h"
#include "Vector.h"
#include "IntervalVector.h"
#include "IntervalMatrix.h"
#include "IntegerMatrix.h"
#include "header_Solve_General_Interval.h"
#include "header_Utilities_Interval.h"
double z;
//F,J,H are the equation, jacobian and hessian procedures
#include "F.C"
#include "J.C"
#include "H.C"

int main()
{
MATRIX BRANCH;
INTERVAL RANGE;
int NUM,i,j,NB_BRANCH;
char texte[400];
double delta=0.05;
double min_delta=1.e-6;
double mindz=1.e-4;
ostream out;

int Nb_Var=2;
int Nb_Eq=2;
Domain(1)=INTERVAL(-100,100);
Domain(2)=INTERVAL(-100,100);
RANGE=INTERVAL(0.2);
NUM=ALIAS_Full_Continuation(Nb_Var,Nb_Eq,F,J,H,Domain,1.e-6,1.e-6,&z,
delta,min_delta,mindz,RANGE,1,BRANCH,&NB_BRANCH);
```

```
//order the branch and write the result in the file
for(i =1;i<=NUM;i++)
     {
       sprintf(texte,"BRANCH%d",i);
       out.open(texte,ios::out);
       for(j =1;j<=NB_BRANCH;j++)
          {
           if(BRANCH(j,Nb_Var+2)!=i)continue;
           out<<BRANCH(j,1)<<" "<<BRANCH(j,2)<<" "<<BRANCH(j,3)<<endl;
          }
      out.close();
      }
}
```

Chapter 10

Integration

10.1 Definite integrals

ALIAS offers various procedures to determine bounds for the value of definite integrals with one or more variables.

10.1.1 Integral with one variable

```
int Integrate(
      INTERVAL_VECTOR (* TheIntervalFunction)
      (int,int,INTERVAL_VECTOR &),
      INTERVAL & TheDomain,
      int Iteration,
      double Accuracy,
      INTERVAL & Result)
```

The simplest integration procedure of ALIAS that should be used only for the simplest function.

- TheIntervalFunction: a procedure in MakeF format to interval evaluate the function
- TheDomain: integration domain
- Iteration: this procedure uses a set of boxes and Iteration is the maximum number of boxes that can be used
- Accuracy: desired accuracy for the integration: the width of the result should be lower than this number
- Result: the range for the integral

This procedure returns 1 if the calculation has been successful, -1 if the desired accuracy cannot be reached and -2 if the number of boxes has been exceeded.

If the function is at least twice differentiable it is possible to use:

```
int IntegrateTrapeze(
          INTERVAL_VECTOR (* TheIntervalFunction)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* SecondDerivative)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL & TheDomain,
          int Iteration,
          double Accuracy,
          INTERVAL & Result)

int IntegrateRectangle(
```

```
          INTERVAL_VECTOR (* TheIntervalFunction)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* SecondDerivative)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL & TheDomain,
          int Iteration,
          double Accuracy,
          INTERVAL & Result)
```

The procedure `SecondDerivative` in `MakeF` format allows to interval evaluate the second derivative of the function.

Alternatively it is possible to use:

```
int IntegrateTaylor(
          INTERVAL_VECTOR (* CoeffTaylor)
          (int,int,INTERVAL_VECTOR &),
          int Order,
          INTERVAL & TheDomain,
          int Iteration,
          double Accuracy,
          INTERVAL & Result)
```

The procedure `CoeffTaylor`, in `MakeF` format, should provide the interval evaluation of the Taylor coefficients of the function up to the order `Order`+1 (i.e. `Order`+2) coefficients).

10.1.2 Integral with multiple variable

`ALIAS` offers 4 procedures to compute definite integral with more than one variable.

```
int IntegrateMultiRectangle(
          INTERVAL_VECTOR (* Function)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* Second_Derivative)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR & TheDomain,
          int Iteration,
          double Accuracy,
          INTERVAL & Result)

int IntegrateMultiRectangle(
          INTERVAL_VECTOR (* Function)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* Second_Derivative)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR & TheDomain,
          int Iteration,
          double Accuracy,
          INTERVAL & Result)
```

In the second form `Gradient` is a procedure in `MakeJ` format that allows to compute the derivatives of the second derivatives of the function. `ALIAS` offer also procedure based on Taylor expansion of the function.

```
int IntegrateMultiTaylor(
          INTERVAL_VECTOR (* CoeffInt)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* RestTaylor)
```

```
          (int,int,INTERVAL_VECTOR &),
          INTEGER_MATRIX &APOWERINT,
          INTEGER_MATRIX &APOWERREM,
          int nbrem,
          int Order,
          INTERVAL_VECTOR & TheDomain,
          int Iteration,
          double Accuracy,
          INTERVAL & Result)
int IntegrateMultiTaylor(
          INTERVAL_VECTOR (* CoeffInt)
          (int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR (* RestTaylor)
          (int,int,INTERVAL_VECTOR &),
          INTEGER_MATRIX &APOWERINT,
          int Order,
          INTERVAL_VECTOR & TheDomain,
          int Iteration,
          double Accuracy,
          INTERVAL & Result)
```

The Taylor expansion of the function F may be written as:

$$F(x_1,\ldots,x_n)=\sum C_{i_1 i_2\ldots i_n}(x_1-h_1)^{i_1}(x_2-h_2)^{i_2}\ldots(x_n-h_n)^{i_n}+R$$

- `CoeffInt`: a procedure in `MakeF` format that compute the coefficients $C_{i_1 i_2\ldots i_n}$
- `APOWERINT`: a table with the exponent $i_1,\ldots,i_n$ for each term
- `RestTaylor, APOWERREM`: in the first form they play the same role than `CoeffInt`, `APOWERINT` for the remainder. There is `nbrem` terms in the remainder
- `RestTaylor`: in the second form a procedure in `MakeF` format that compute the remainder
- `Order`: the degree of the remainder

Chapter 11

Miscellaneous procedures

11.1 Management of boxes list

11.1.1 Adding boxes to a list

The procedure `ALIAS_Add_Box` allows to add boxes to a list:

```
int ALIAS_Add_Box(int DimVar,INTERVAL_MATRIX &Box,int *Current_Box,int *Nb_Box,
        INTERVAL_MATRIX New_Box, int Nb_New_Box,int MaxBox)
```

where

- `DimVar`: number of ranges in the box
- `Box`: the list of box
- `Current_Box`: the insertion point for the new boxes
- `Nb_Box`: the current number of boxes in the list. Afterward the new number of boxes in the list
- `New_Box`: the list of boxes to add
- `Nb_New_Box`: the number of boxes to add
- `MaxBox`: the maximal number of boxes in the list

If `Reverse_Storage` is set to 0 the new boxes will be added at the end of the list. Otherwise all the boxes to add that are smaller (i.e. have a smaller width or if the width are identical a smaller mean diameter) than the box in position `Current_Box`+1 in the list are added to the list starting at position `Current_Box`.

There is a similar procedure if the simplified jacobian is also stored:

```
int ALIAS_Add_Box_Gradient(int DimVar,int DimEq,INTERVAL_MATRIX &Box,
             INTEGER_MATRIX &GBox,
             int *Current_Box,int *Nb_Box,
             INTERVAL_MATRIX New_Box,
             INTEGER_MATRIX GNew_Box,int Nb_New_Box,int MaxBox)
```

where

- `GBox`: the list of simplified jacobian
- `GNew_Box`: the simplified jacobian of the new boxes

If the jacobian is stored instead of the simplified jacobian you may use

```
int ALIAS_Add_Box_Hessian(int DimVar,int DimEq,INTERVAL_MATRIX &Box,
                   INTERVAL_MATRIX &GBox,
                   int *Current_Box,int *Nb_Box,
                   INTERVAL_MATRIX New_Box,
                   INTERVAL_MATRIX GNew_Box,int Nb_New_Box,int MaxBox)
```

11.1.2 Freeing boxes in a list

An algorithm is currently processing box `Current` in the list: hence box from 1 to `Current` -1 are free. The procedure

```
void ALIAS_Free_Storage(int *Current,int *nb_box,int DimVar,int DimEq,
             INTERVAL_MATRIX &Box,INTERVAL_MATRIX &GBox)
```

allows to free this unused storage by shifting: the box initially at position `Current` will become box number 1.

11.2 Void procedures

In some cases the routine of ALIAS-C++ have procedure arguments that are useless. ALIAS-C++ proposes void procedures that may be used as such argument and have no effect on the calculation:

- `int ALIAS_Simp_Proc_Void`: a simplification procedure that always return 0
- `INTERVAL_MATRIX ALIAS_Gradient_Void(int l1,int l2,INTERVAL_VECTOR &P)`: a gradient procedure that returns a not defined interval matrix
- `int ALIAS_Matrix_Void(INTERVAL_VECTOR &Box,INTERVAL_MATRIX &P)`: a procedure that may be used for routine asking for a matrix but the return code is 0, stating that the matrix has not been computed
- `INTERVAL_MATRIX Hessian_Void(int i1, int i2, INTERVAL_VECTOR &X)`: a procedure that returns an empty (1,1) interval matrix
- `void Compute_Gradient_Non_Linear_Void(INTERVAL_VECTOR &X,INTERVAL_MATRIX &J)`: a procedure for the simplex algorithm that is empty
- `INTERVAL_VECTOR ALIAS_Void_Update_Range(double z,INTERVAL_VECTOR &Input)`: a `Range` procedure for the continuation algorithm that just returns `Input`

11.3 Bisection procedures

We indicate here the program used for the bisection in the general solving procedures with the abbreviation SG `Solve_General_Interval`, SGG for `Solve_General_Gradient_Interval` and SJH for `Solve_General_JH_Interval`. All procedures return the number of the variable that will be bisected (a return code of 0 or lower is an error message). .

SG, mode 1, 3, 4, SGG, SJH, mode 1:

```
int Select_Best_Direction_Interval_Fast(int Dim,INTERVAL_VECTOR &Input)
```

where `Dim` is the number of unknowns and `Input` is the box

SG, mode 2, SGG, mode 6:

```
int Select_Best_Direction_Interval(int Dim,int Dimension_Eq,int Order,
          INTERVAL_VECTOR (*Func)(int,int,INTERVAL_VECTOR &),
          INTERVAL_VECTOR &Input)
```

where `Dimension_Eq` is the number of equations and `Order` the current ordering mode of the boxes.

SGG, SJH, mode 2:

```
int Select_Best_Direction_Smear(int Dim,int Dimension_Eq,
                INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                INTERVAL_VECTOR &Input)
```

where `Func` is the name of the procedure that interval evaluates the equations and `Gradient` the procedure that evaluates the derivative of the equations.

SGG, SJH, mode 3:

```
int Select_Best_Direction_Smear_Bounded(int Dim,int Dimension_Eq,
              double Bound,
              INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
              INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
              INTERVAL_VECTOR &Input)
```

where `Bound` represents `ALIAS_Bound_Smear`.

SGG, SJH, mode 4:

```
int Select_Best_Direction_Gradient_Interval(int Dim,int Dimension_Eq,
                INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                INTERVAL_VECTOR &Input, double *SINDEX)
```

where `SINDEX` is the criteria value.

SG, SGG, SJH, mode 5:

```
int Select_Best_Direction_Interval_Round_Robin(int Dim,INTERVAL_VECTOR &Input)
```

SG, mode 6:

```
int Select_Best_Direction_Grad(int Dim,int Dimension_Eq,
                   INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                   INTERVAL_VECTOR &Input)
```

SJH, mode 6:

```
int Select_Best_Direction_Weight(int DimVar,INTERVAL_VECTOR &Input)
```

SGG, mode 7:

```
int Select_Best_Direction_Interval_Proc(int Dim,int Dimension_Eq,
                int Nb_Var,int Order,
                INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                INTERVAL_VECTOR &Input,int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where `Nb_Var` is set to 1.

We have a specific bisection procedure for the distance equations that corresponds to mode 1 of SG:

```
int Select_Best_Direction_Distance(int Dimension,int Dimension_Eq,
                 INTEGER_MATRIX &APOW,
                 MATRIX &ACONS,VECTOR &LI,
                 int NB_EQV,int NB_VARV,MATRIX &AVARV,
                 INTERVAL_VECTOR &Input,double *SPLIT)
```

where `SPLIT` is the bisection point.

A specific version of the smear procedure is used for determining the condition number of a matrix (mode 2):

```
int Select_Best_Direction_Smear_CN(int Dim,int Dimension_Eq,
                 INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                 INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                 INTERVAL_VECTOR &Input)
```

Similarly there is a special mode 2 for the procedures dealing the with the eigenvalues of a matrix:

```
int Select_Best_Direction_Smear_Eigen(int Dim,int Dimension_Eq,
                  INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                  INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                  INTERVAL_VECTOR &Input)
```

11.4 3B procedures

The 3B procedures are implemented according to the solving algorithms. For `Solve_General_Interval`:

```
int ALIAS_3B_Normal(int Dimension,int DimEq,
                   INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                   INTERVAL_VECTOR &Input,int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where

- `Dimension`: number of unknowns
- `DimEq`: number of equations
- `Func`: procedure to evaluate the equations
- `Input`: the current box
- `Simp_Proc`: an optional simplification procedure

For `Solve_General_Gradient_Interval`:

```
int ALIAS_3B_Gradient(int DimVar,int DimEq,
                   INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                   INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                   INTERVAL_VECTOR &Input,INTEGER_VECTOR &A1,
                   int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where`Gradient` is the name of the procedure that evaluate the derivatives of the equations
For `Solve_General_JH_Interval`:

```
int ALIAS_3B_Hessian(int DimVar,int DimEq,
                   INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                   INTERVAL_MATRIX (* Gradient)(int, int, INTERVAL_VECTOR &),
                   INTERVAL_MATRIX (* Hessian)(int, int, INTERVAL_VECTOR &),
                   INTERVAL_VECTOR &Input,INTERVAL_MATRIX &A1,
                   int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where `Hessian` is the procedure that computes the Hessian of the equations.
A specific 3B exists for distance equations:

```
int ALIAS_3B_Distance(int Dim,int Dimension_Eq,
                  INTEGER_VECTOR &VAR_TOUCHED,
                  INTEGER_MATRIX &APOW,MATRIX &ACONS,
                  VECTOR &LI,int NB_EQV,int NB_VARV,MATRIX &AVARV,
                  INTERVAL_VECTOR &Input,int (* Simp_Proc)(INTERVAL_VECTOR &))
```

For the eigenvalue of matrices we have:

```
int ALIAS_3B_EigenValues(int Dimension,int Dimension_Eq,int Degree,
                    int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
                    int Has_Matrix,
                    INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                    int *Has_Gradient,
                    INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
                    INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
                   int Nb_Points,int Use_Solve,
                   double Accuracy,double Accuracy_Var,
                    INTERVAL_VECTOR &Input,
                    int (* Solve_Poly)(double *, int *,double *),
                    int (* Simp_Proc)(INTERVAL_VECTOR &))
```

For the geometry of region with given eigenvalue range we have:

```
int ALIAS_3B_Min_Max_EigenValues_Area(int Degree,int Nb_Parameter,
                    INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
                    INTERVAL_VECTOR (* TheCoeffCentered)(INTERVAL_VECTOR &,double),
                    int Nb_Constraints,INTEGER_VECTOR &Type_Eq,
                    int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
                    int Has_Matrix,
                    INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                    int *Has_Gradient,
                    INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
                    double *Seuil_First_Sol1,int RealRoot,
                    int (* Solve_Poly)(double *, int *,double *),
                    INTERVAL_VECTOR &Input,
                    int (* Simp_Proc)(INTERVAL_VECTOR &))
```

For the optimum of the condition number of matrices we have:

```
int ALIAS_3B_CN(int Dimension,int Dimension_Eq,int Degree,
                    int (* TheMatrix)(INTERVAL_VECTOR &, INTERVAL_MATRIX &),
                    int Has_Matrix,
                    INTERVAL_VECTOR (* Func)(int,int,INTERVAL_VECTOR &),
                    int Has_Gradient,
                    INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
                    INTERVAL_VECTOR (* TheCoeff)(INTERVAL_VECTOR &),
                    int Nb_Points,int Absolute,
                    double Accuracy,double Accuracy_Var,
                    INTERVAL_VECTOR &Input,
                    int (* Solve_Poly)(double *, int *,double *),
                    int (* Simp_Proc)(INTERVAL_VECTOR &))
```

In the procedure `ALIAS_Geometry_Carre` we use:

```
int ALIAS_3B_Constraints(int Nb_Parameter,int Dimension_Eq,
                    INTEGER_VECTOR &Type_Eq,INTEGER_VECTOR &Imperatif,
                    int Degree,double Accuracy_Variable,double Accuracy,
                    double Accuracy_Geometry,
                    INTERVAL_VECTOR (* F)(int,int,INTERVAL_VECTOR &),
                    int *Has_Gradient,
                    INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
                    INTERVAL_VECTOR &Input,
                    int (* Simp_Proc)(INTERVAL_VECTOR &))
```

11.5 Volume of a box

The procedure `ALIAS_Volume` allows to compute the volume of a box. It has 2 syntax:

```
double ALIAS_Volume(int Dim,INTERVAL_VECTOR &Box)
double ALIAS_Volume(int Dim,int box,INTERVAL_MATRIX &BOX)
```

The first syntax allows to compute the volume of the interval vector `Box` of size `Dim`, while the second syntax allows to compute the volume of the box defined by the `box`-th row of the interval matrix `BOX`: the row is supposed to have `Dim` elements.

11.6 Filtering

11.6.1 Square adjustment

Assume that an equation is written as $(x-W)^2 = U$ where U, W are intervals and x lie in the interval [LOW, HIGH]. According to the interval values LOW and HIGH may not be compatible with the equation. The procedure ALIAS_Ajuste_Square allows one to obtain new bounds for the range for x:

```
int ALIAS_Ajuste_Square(INTERVAL &U,INTERVAL &W,double *LOW,double *HIGH)
```

This procedure will return:

- -1 if x cannot be a solution of the equation
- 1 if new values have been found for LOW and/or HIGH
- 2 if the sum of the absolute value of the change between the initial value of LOW, HIGH and their new value is greater than ALIAS_Seuil_Ajuste or if this sum is larger than the initial diameter of x multiplied by ALIAS_Seuil_Ajuste_Percent divided by 100
- 0 otherwise

11.7 Box reduction

Let Sol be a solution of a system that has been found in the box B while the algorithm was processing a list of boxes BOX. It is of interest to treat the box and the list of boxes in such way that Sol is no more included in the set of boxes i.e. to have in the list of boxes only the complementary of each box with respect to the box Sol. Note that for a problem with n unknowns this complementary may be described by up to $2n$ boxes.

This may be obtained by using:

```
int Sol_Reduction(int Dimension,int Dimension_Eq,
                 INTERVAL_MATRIX &BOX,
                INTERVAL_VECTOR &B, INTERVAL_VECTOR &Sol,
                int *nb,int type,
                int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where

- nb: the total number of boxes in the list BOX
- type: allow to create at most type new boxes that will be appended to BOX
- Simp_Proc: a simplification procedure for the system

This algorithm will return -1 if the solution box covers completely the box B. Note that this algorithm may create new boxes but only ALIAS_Allows_N_New_Boxes are allowed to be created (with a default value of 2). A similar procedure exists when gradient boxes have also to be stored:

```
int Sol_Reduction(int Dimension_Var,int Dimension_Eq,
  INTERVAL_MATRIX &BBOX,
  INTERVAL_MATRIX &GBBOX,
  INTERVAL_MATRIX (* Gradient)(int, int,INTERVAL_VECTOR &),
  INTERVAL_VECTOR &P, INTERVAL_VECTOR &PP1,
  int current_box,int *nb_box,
  int nb_max_box,int type,
  int (* Simp_Proc)(INTERVAL_VECTOR &))
```

where

- `GBBOX`: the gradient boxes
- `current_box`: the number of the current box
- `nb_box`: the total number of boxes
- `nb_max_box`: the maximal number of new boxes that can be created
- `type`: if `nb_max_box` is lower than 2`Dimension_Var` then the full complementary boxes cannot be generated. Hence the boxes that will be generated will still have an intersection with `PP1`. But the set of generated boxes will depend on the order in which the variables are considered when generating the new boxes. `type` allows one to specify this order. If set to 1 the algorithm will consider the variables according to the width of their range, the largest first. If set to 2 it will consider the variable according to their influence in the sense of the smear function (see section 2.4.1.3. For any other value the variables will be considered in the order of their definition.

Note that in the `Solve_General_Gradient_Interval` and `Solve_General_JH_Interval` we use for `nb_max_box` the value of `ALIAS_Allows_N_New_Boxes` (with a default value of 2) and for `type` the value of `ALIAS_Type_N_New_Boxes` (default value of 0)

11.8 Binomial

The calculation of C_n^m may be done with the procedure

```
int Binomial(int n,int m)
```

Chapter 12

Parser, Generic Solver and Analyzer

12.1 The ALIAS parser

As it has been seen in the previous chapters the interval evaluation of an equation is highly dependent upon its formulation (for example the evaluation of x^2+x may be quite different from the evaluation of $x(x+1)$ for the same range on x). For testing purposes it may therefore be interesting to change the formulation of an equation (for example by using MAPLE), write its analytical form in a file and test what will be the interval evaluation of this equation in this analytical form: this is one objective of the `ALIAS` parser. Basically this parser takes as input:

- a file, called a *formula file*, in which is written the analytical form of the equation
- name and ranges for the variables

and will then produces as output the interval evaluation of the equations. In the formula file you may indicate an arbitrary number of equations, each of them being prefixed by `eq=`. For example

```
eq=(y^2-1)*z+(2*y*t-2)*x
eq=2+(-10*t+(-10+2*y*t)*y)*y+((4+4*y^2)*z+(4+4*y*t-x*z)*x)*x
eq=(2*y*t-2)*z+(t^2-1)*x
eq=2+(4*x+(4-x*z)*z)*z+((-10+4*z^2)*y+(-10+4*x*z+2*y*t)*t)*t
```

is a valid formula file in *parser format*. There is however a limitation on the number of unknowns which cannot exceed 200. Note that the variable names in a formula file may be any name composed with lower and upper cases, integer numbers and underscore, with the constraint that the first character should be a letter. Note also that a MAPLE library enables to produce a formula file directly from MAPLE equations (see section 12.3).

The parser may handle almost any complex analytical equation based on the most classical mathematical functions, using MAPLE notation. Currently you may use the following operators:

- arithmetic operator: "+","-","/","*","^ " or "**" (power)
- trigonometric and inverse trigonometric functions: `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan` (which may be used with the syntax `arctan`(x) or `arctan`(y,x))
- hyperbolic functions: `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, `arctanh`
- mathematical functions: `abs` (absolute value), `log`, `log10`, `exp`, `sqrt`
- integer functions: `ceil` (returns an interval which bounds are the smallest integer greater than or equal to each bound of the box, `floor` (returns an interval which bounds are the greatest integer less than or equal to each bound of the box), `round` (returns an interval which bounds are the nearest integer of each bound of the box)

Beside these functions we have added a few more operators:

- `Min`: this operator takes as input a list of mathematical expression and will return an interval reduced to the minimum value of all the lower bound of the interval evaluation of each term in the list. For example:

  ```
  Min(3,3*x1,3*x2+x3,4*(x2+x1))
  ```

  for `x1` in [-1,1], `x2` in [0,2], `x3` in [3,10] will return the interval [-4,-4]

- `Max`: this operator is similar to `Min` except it returns the maximal value of the upper bound on the interval evaluation. For the previous example `Max` will return the interval [16,16]

- `MinMax`: this operator applied on the list $\mathcal{L}$ will return the interval [`Min`($\mathcal{L}$), `Max`($\mathcal{L}$)]. In the previous example the interval will be [-4,16]

The parser may also handle intervals. For example you may evaluate an equation in which some coefficients are intervals. In the equation these coefficients should be indicated using the MAPLE notation as in the following example:

```
INTERVAL(0.1 .. sin(1))
```

12.1.1 Using the ALIAS parser in a program

The formula file name will be used as an input in all the procedures described in the following sections. As for the variable you must fill the text variable `variable_name_IS` defined by

```
char variable_name_IS[200][200];
```

with the name of you variable and set the integer variable `Unknowns_Number_IS` to the number of unknowns in your equations as in

```
strcpy(variable_name_IS[0],"x");
strcpy(variable_name_IS[1],"y");
Unknowns_Number_IS=2;
```

Note that the variable names may be any name composed with lower and upper cases, integer numbers and underscore, with the constraint that the first character should be a letter.

In the following procedures we will then use an interval vector to define the ranges for the unknowns. In this vector the ranges are ordered so that this order will correspond to the order in the declaration of the variable name (in the previous example the first range will be the range for `x` and the second range will be the range for `y`).

12.1.1.1 Evaluating a single formula

An equation may be evaluated by using the procedure `Evaluate_Interval_With_Parser` with the following syntax:

```
int Evaluate_Interval_With_Parser(
        char *texte,INTERVAL_VECTOR &P,int Unknowns,INTERVAL &Value)
```

The parameters are:

- `texte`: the name of the file describing the analytical form of the equation
- `P`: the ranges for the unknowns
- `Unknowns`: the number of unknowns
- `Value`: the interval evaluation of the equation

This procedure returns 1 if the equation has been successfully evaluated, 0 otherwise.

12.1.1.2 Evaluating multiple equations

Using the parser you may evaluate more than one equation described in a single formula file. Each equation in the file should be prefixed by the symbol `eq=` like in:

```
eq=-x1^2-x2+x3
eq=x1^2-x2+x3
eq=round(x3)
```

All equations in a system of equations may be evaluated by using the procedure `Evaluate_Interval_With_Parser` with the following syntax:

```
int Evaluate_Interval_With_Parser(
        char *texte,INTERVAL_VECTOR &P,int Unknowns,int NbEq,INTERVAL_VECTOR &Value)
```

The parameters are:

- `texte`: the name of the file describing the analytical form of the equation
- `P`: the ranges for the unknowns
- `Unknowns`: the number of unknowns
- `NbEq`: the number of equations
- `Value`: the interval evaluation of the equations

This procedure returns 1 if the equations have been successfully evaluated, 0 or a negative number otherwise.

Thus for evaluating the formula described by the formula file `toto`:

```
eq=-x1^2-x2+x3
eq=x1^2-x2+x3
eq=round(x3)
```

for the ranges `x1` in [-1,1], `x2` in [0,2], `x3` in [1.1,2.3] you may use the following piece of code:

```
INTERVAL_VECTOR P(3);
INTERVAL_VECTOR F(3);
strcpy(variable_name_IS[0],"x1");
strcpy(variable_name_IS[1],"x2");
strcpy(variable_name_IS[1],"x3");
Unknowns_Number_IS=3;

P(1)=INTERVAL(-1,1);//range for x1
P(2)=INTERVAL(0,2);//range for x2
P(3)=INTERVAL(1.1,2.3);//range for x3

Evaluate_Interval_With_Parser("toto",P,3,3,F);
cout<<F<<endl;//interval evaluation of the 3 equations
```

Instead of evaluating the whole system of equations we may have to evaluate one or more particular equation(s) of the system. The following procedures enable to evaluate efficiently particular equations in a system but before using them we need to do some initialization. This is done by calling once the procedure `Init_Evaluate_Equations_With_Parser` with the syntax:

```
int Init_Evaluate_Equations_With_Parser(char *texte,int *EqStart)
```

The parameters are:

- `texte`: the name of the formula file
- `EqStart`: an array of integer with size equal to the number of equations in the formula file

This procedure returns the number of equations of the system described in the formula file and -1 if the formula file does not exist. It must be used on each formula file with one particular `CharEq` per file.

After this initialization the following procedure may be used to evaluate one particular equation in a system:

```
int Evaluate_Equations_With_Parser(char *texte,INTERVAL_VECTOR &P,int Unknowns,
                int NbEq,int EqToRead,int *EqStart,INTERVAL &Value)
```

The parameters are:

- `texte`: the name of the file describing the analytical form of the equation
- `P`: the ranges for the unknowns
- `Unknowns`: the number of unknowns
- `NbEq`: the number of equations
- `EqToRead`: the number of the equation that has to be evaluated (the equation number start at 1)
- `EqStart`: the array of integer produced by the initialization procedure for the formula file
- `Value`: the interval evaluation of the equation `EqToRead`

This procedure will return 1 on a successful completion. It returns 0 if there was a problem in the evaluation and -1 if the numbering of the equations as indicated in `EqToRead` is not correct. We may also evaluate more than one equation in a system with:

```
int Evaluate_Equations_With_Parser(char *texte,INTERVAL_VECTOR &P,int Unknowns,
                int NbEq,int EqToRead1,int EqToRead2,int *EqStart,INTERVAL &Value)
```

The parameters are:

- `texte`: the name of the file describing the analytical form of the equation
- `P`: the ranges for the unknowns
- `Unknowns`: the number of unknowns
- `NbEq`: the number of equations
- `EqToRead1`: the number of the first equation that has to be evaluated (the equation number start at 1)
- `EqToRead2`: the number of the last equation that has to be evaluated
- `EqStart`: the array of integer produced by the initialization procedure for the formula file
- `Value`: the interval evaluation of the equation `EqToRead`

This procedure will return 1 on a successful completion. It returns 0 if there was a problem in the evaluation and -1 if the numbering of the equations as indicated in `EqToRead1` or `EqToRead2` is not correct. We may also use the gradient of the equations to improve the evaluation. The syntax is:

```
int Evaluate_Equations_With_Parser_Gradient(int Unknowns,int NbEq,char *texte,
                   char *gradient,INTERVAL_VECTOR &P, INTERVAL_VECTOR &Value,int Exact)
```

The parameters are:

- `Unknowns`: the number of unknowns
- `NbEq`: the number of equations
- `texte`: the name of the file describing the analytical form of the equation
- `gradient`: the name of the parser file giving the gradient matrix of the equations. The order in this file is to define successively the derivative of the first equation with respect to each variable, then the derivative of the second equation and so on
- `P`: the ranges for the unknowns
- `Value`: the interval evaluation of the `NbEq` equations
- `Exact`: if 0 stop processing the equations as soon as the interval evaluation of the n-th equation does not include 0 (the interval evaluation of the remaining n,..,`NbEq` equations will be set to 1. If 1 the procedure computes the interval evaluation of all the equations

12.1.2 Example of use of the parser

The program `Evaluation` provided in the `ALIAS` distribution is a simple example of the use of the evaluation through the parser. In its simplest form it takes as first argument a formula file and as second argument a *range file* (see section 12.2 for the definition of a range file) and print the interval evaluation of all the equations:

```
Evaluation [formula file] [range file]
```

A third integer argument `n1` may be the number of the equation we want to evaluate (the number start at 1), while if a fourth integer argument `n2` is present the evaluation of the equations from `n1` to `n2` will be printed.

12.2 The generic Solver

A generic solver, called `Interval_Solver`, based on the parser is available in the `ALIAS` distribution. The basic arguments of this program is a formula file which defines the equations to be solved and a *range file* which describe the possible ranges for each of the unknowns in the equations. A line of this range file starts by the name of the variable, followed by two numbers which indicate the range for the variable. For example the following lines define a valid range file

```
psi 0 0.2
x 1 2.5
```

If `f` denotes a formula file and `r` a range file then you may run:

```
Interval_Solver -F f -R r
```

In that case the program will call the procedure `Solve_General_Interval`[1] of the `ALIAS` library.

Instead of using the `Solve_General_Interval` procedure of `ALIAS` you may use the `Solve_General_Gradient_Interval` procedure which requires the gradient of the equations. The gradient equations are defined in a formula file (e.g. `g`) and you may then run:

```
Interval_Solver -F f -G g -R r
```

Note that the equations in the gradient formula file should be ordered with respect to the unknowns using the same ordering than the one used in the range file. Thus for the range file:

```
x 1 2.5
y 0 0.2
```

and the formula file `f`:

```
eq=y+3*x
eq=x^2+x*y
```

denoted `eq1, eq2`, the first line of the gradient formula file should be d`eq1`/dx, the second line d`eq1`/dy and so on, which will lead to the following gradient file:

```
eq=3
eq=1
eq=2*x+y
eq=x
```

Similarly, instead of using the `Solve_General_Gradient_Interval` procedure of `ALIAS` you may use the `Solve_General_JH_Interval` procedure which requires the gradient and hessian of the equations. The hessian equations are defined in a formula file (e.g. `h`) and you may then run:

```
Interval_Solver -F f -G g -H h -R r
```

[1] Well, at this time not exactly the same program is used and the solving algorithms may not be as efficient as the one in the C++ library

Note that the hessian formula file should respect the same ordering than for the gradient formula file. For our previous example the hessian file will be:

```
eq=0
eq=0
eq=0
eq=0
eq=2
eq=1
eq=1
eq=0
```

Note also that the various formula files may be easily obtained by using the MAPLE procedures described in the next section (12.3).

The result of the computation will be written in a *result file*. Each line of the result file gives the range for one of the unknowns, using the same ordering than for the range file (in our example the range of `x`, followed by the range for `y`). A range is indicated by two reals separated by a blanc character. If instead of range the result file contain the keyword `FAILED` it means that the solver has been unable to solve the equations. This keyword is followed by an integer which indicate the reason of the failure:

- `-1`: an insufficient storage space has been defined for the solver
- `-4`: the number of equation in the formula file is incorrect

The default name for the result file is `.result_IS`.

Whatever the used solving procedure, some parameters are to be set. The solver will prompt you to give the values of these parameters. If you want to avoid this interactive behavior you may define the parameters in a *configuration file* and the solver will proceed directly to the solution. A configuration file consists in a list of keywords followed by a value. Most of these keywords refers to parameters in the solving procedure of `ALIAS`, please refer to the corresponding section for their meaning:

- `Result_File name`: the result will be stored in the `name` file instead of the default `.result_IS` file.
- `Iteration integer`: the maximum number of boxes that may be used by the algorithm
- `Debug integer`: the debug level for the solving procedure as indicated by `Debug_Level_Solve_General`
- `AccuracyF real`: the `epsilonf` parameter of the solving procedure
- `AccuracyV real`: the `epsilon` parameter of the solving procedure
- `Order integer`: the `Order` parameter of the solving procedure
- `Stop integer`: the `Stop` parameter of the solving procedure
- `MaxSol integer`: the maximum number of solutions you are looking for
- `Kanto integer`: the `ApplyKanto` parameter of the solving procedure
- `DifferenceS real`: the `Dist` parameter of the solving procedure
- `Machine_Name char`: the name of a computer that will be used during a run of the parallel version of the solver
- `Single_Bisection`: if you use the solver with the -H or -G option the bisection process will bisect only one variable at a time

12.2.1 Dealing with inequalities

The generic solver may deal also with inequalities. Here we will add an argument to the generic solver whose syntax will be

```
-I [inequalities file]
```

`inequalities file` is a formula file, called the *inequalities file* that describe equations in the same manner than a formula file. But the solver will output as solutions only the ranges for which all the equations in the inequalities file may be positive or equal to 0 and eliminate the solutions ranges for which at least one equation is strictly negative. Note that the algorithm stops only according to the values of the equations or the diameter of the ranges and not according to the values of the inequalities.

12.2.2 Dealing with parametric system

It may also be interesting to investigate the case of parametric systems, i.e. of systems for which some coefficients will be described in analytical form in the formula file, but which are not unknowns. An additional argument for the generic solver will be a file, called the *parameter file* which will give specific values for the parameter. For example in the equation:

```
eq1:=x^2-x*cos(y)-x1:
```

`x1` may be considered as a parameter while `x, y` are the unknowns. Each line of the parameter file indicates a name of a parameter and its value. Thus

```
x1 1
```

is a valid parameter file. Then you may use the programs of the MAPLE library (see section 12.3) to generate the parser files for the solver: these programs enable to specify that `x1` is not an unknown. The generated files will contain the parameter `x1` and if you want to solve a particular occurrence of the parametric system you just have to add an argument to the command line of the generic solver:

```
-P [parameter file]
```

and if you want to study another occurrence (for example when the value of `x1` is 2) you just have to change the value in the parameter file.

Some specific parameters with special names may also be used to speed up the computation, see section 12.4.4.2.

12.3 MAPLE library for the Interval Solver

All these procedures are available in The `lib_IS.m` library contains various procedures that may be useful for producing files for the parser. The following MAPLE programs are available:

- `cout(eq)`: estimates the cost of the computation of `eq` in terms of number of arithmetic operations. This a rather rough approximation: it just compute the total number of addition, multiplication and function calls without any weight between these operators.

- `minimal_cout(eq)`: returns an expression equivalent to `eq` but with a minimal cost of computation. Basically it consider the expanded form of the equation and compute its computational cost using the `cout` procedure. Then it consider the Horner forms of the equation using all possible ordering of the unknowns (if you have up to 6 unknowns). The cost of these forms are computed and as soon as a form as a lower cost then it is stored as the potential return for the function. If two forms have equal cost, then the form with the lower number of operations (obtained through the `nops` procedure) is retained.

- `write_equation`: this procedure takes as argument an equation or a list of equations and a name. The equation will be written in a minimal form in the formula file defined by the name

- `write_gradient_equation`: this procedures takes as as argument an equation or a list of equations and a name. The gradient of the equations in minimal form will be written in the formula gradient file defined by the name. If no third argument is provided the ordering of the gradient matrix with respect to the unknowns will be the one provided by the MAPLE procedure `indets`. A third argument may be provided and should be a list of unknown names which will be used for the ordering of the gradient matrix (i.e. if the list is `[x1,x2,x3]` for the equation `eq` the first equation will be `deq/dx1`, the second `deq/dx2`..). A fourth argument may be the list of unknowns and is used for parametric system for which the number of undefined variables may be larger than the number of unknowns (otherwise the procedure will return an error message indicating that the equation has more indeterminate than the number of unknowns given in the third argument).

- `write_hessian_equation`: a similar procedure than for the gradient except it computes the hessian matrix of the equations

An example of the use of this library follows:

```
readlib(lib_IS):
eq1:=y**2*z+2*x*y*t-2*x-z:
eq2:=-x**3*z+4*x*y**2*z+4*x**2*y*t+2*y**3*t+4*x**2-10*y**2+4*x*z-10*y*t+2:
eq3:=2*y*z*t+x*t**2-x-2*z:
eq4:=-x*z**3+4*y*z**2*t+4*x*z*t**2+2*y*t**3+4*x*z+4*z**2-10*y*t-10*t**2+2:
write_equation([eq1,eq2,eq3,eq4],"fcap"):
write_gradient_equation([eq1,eq2,eq3,eq4],"gcap",[x,y,z,t]):
write_hessian_equation([eq1,eq2,eq3,eq4],"hcap",[x,y,z,t]):
```

The formula file for the 4 equations will be written in the file `fcap` while the jacobian and hessian will be written in the file `gcap, hcap`. Note that the range file should define the ranges for the unknowns in the order `x,y,z,t`.

12.4 The generic analyzer

12.4.1 Principle

The purpose of the generic analyzer is to try to refine a box and to propose smaller sub-intervals which may contain real roots of the system of equations (which may be algebraic or not). Two types of tools are used for that purpose:

- as soon as one of the equation in the system is algebraic in at least one of the unknowns then the tools described for determining ranges on the roots of an univariate polynomial with interval coefficients are used recursively

- otherwise Kantorovitch theorem is used (if you have m equations and n unknowns with $m > n$, then this theorem is applied only for the first n equations)

More precisely consider the system in two unknowns:

$$x^2 + x\cos(y) + 0.5 = 0$$
$$y^2 - y + \sin(x) - 0.1 = 0$$

In a first step the first equation is considered: it will be considered as a polynomial in x with the interval coefficients $1, \cos(y), 0.5$. Bounds on the real roots of this polynomial will be computed and the range on x will be updated if necessary. Then the second equation will be considered as a polynomial in y with interval coefficients $1, -1, \sin(x), -0.1$ Bounds on the real roots of this polynomial will be computed and the range on y will be updated if necessary. After these two steps if a range has been improved we will reiterate the process until no further improvement is obtained.

We will then try to improve the obtained ranges by using Kantorovitch theorem. We consider the middle point of the ranges and test if Kantorovitch is able to determine a ball centered at this point for which Newton method will converge toward the unique solution in the ball. If the answer is positive we will use Newton with as starting point the center of the ball: the result, a point, is stored as one of the result ranges. The intersection of the ball with the initial range will then be computed and if non empty will be stored for further processing.

We may indeed perform an in-depth analysis. The previous process is applied on the initial range and will lead to a set of ranges (possibly to the same ranges than the initial one); we may then bisect the obtained ranges and reiterate the process on the resulting ranges, discarding those for which the interval evaluation for at least one of the equations does not include 0. We define a *depth level* for the algorithm as the number of bisection steps that will be used in the process (a depth 0 means that no bisection will be done).

12.4.2 Practical implementation

12.4.2.1 The generic analyzer

The purpose of the generic analyzer is to consider as **input** any system defined in a MAPLE file and an initial set of ranges for the unknowns and to produce as **output** a set of ranges, strictly included in the initial set, that may contain real roots of the system. The output is presented as a file that can be used directly as input for the generic solver.

A formal-numerical approach has been used to develop this generic analyzer. Basically it takes as input a MAPLE file, called the *formula file*, which define the analytical form of the equations of the system and a *range file* in in which are defined both the names of the variables and their initial ranges (a range file may contain more than one set of ranges for the unknowns). The convention in the formula file is to define each equation of the system in the MAPLE file as a sentence prefixed by the key-word `eqn` where `n` is the number of the equation starting at 1. For example

```
eq1:=y**2*z+2*x*y*t-2*x-z:
eq2:=-x**3*z+4*x*y**2*z+4*x**2*y*t+2*y**3*t+4*x**2-10*y**2+4*x*z-10*y*t+2:
eq3:=2*y*z*t+x*t**2-x-2*z:
eq4:=-x*z**3+4*y*z**2*t+4*x*z*t**2+2*y*t**3+4*x*z+4*z**2-10*y*t-10*t**2+2:
```

is a valid formula file, describing a set of 4 equations. The range file has as many lines as unknowns and each line start by the name of the variable followed by two numbers indicating the range for the variable. For example

```
x -100 100
y -100 100
z -100 100
t -100 100
```

is a valid range file for the previous formula file. The analyzer is able to deal with infinity in the ranges: instead of using numbers you may indicate `infinity` or `-infinity`. In a first step the analyzer will use MAPLE to produce various intermediary files which are needed for the algorithm to proceed (see section 12.4.4.1). For example it will write in the `/tmp/Equation` file a description of the system compatible with the syntax used by the parser. Similarly you will find in the `/tmp/Gradient_Equation` and `Hessian_Equation` files a description of the gradient and hessian of the system. As soon as all the necessary files have been written, the algorithms described in the previous section will be used.

The result of the analyzer is a file (by default `.result_IA`) that describe the possibles ranges which may contain real roots of the system. This file may then be used as input for the solver. The analyzer will also produced the `.gradient_IA` file that indicate the value of the gradient matrices for each of the out ranges.

The analyzer is run by using the syntax:

```
Interval_Analyzer -F [formula file] [-G] [-H] -R [range file]
```

In this list of argument only the `-F, -R` are compulsory. The `-G option` indicates that you will use the monotonicity of the equations (evaluated through an interval evaluation of the gradient) to improve the interval evaluation of the equations. The `-H` indicates that you will also use Kantorovitch theorem to isolate the real roots of the equations. In that case if you have m equations and n unknowns with $m > n$, then Kantorovitch will be applied on the n first equations.

This program will then ask you the maximum number of ranges that may be created by the algorithm, an accuracy ϵ_f on the value of the equations that will be used to stop the computation of the Newton scheme, a minimal value for the maximal diameter for the output ranges (which is not used right now), a depth level, a maximal diameter for the ranges over which Kantorovitch is not used and the value of the debug level (0 if the program is to be run quietly, 1 to see how many boxes are dealt with and 2 for a full debug). Note that if you are using the `-H` option and have m equations and n unknowns, with $m > n$ and if the algorithm returns roots of the system (i.e. the output ranges are reduced to a point) it will mean that these points are guaranteed roots of the n first equations (you may use the Newton scheme with a lower ϵ_f on these roots and you will still get a root), while the absolute values of the $m-n$ remaining equations at these points will be lower than ϵ_f but there is no guarantee that these points are effectively roots of these equations.

Alternatively you may use a *configuration file* to give values for these parameters.

A configuration file consists in a list of keywords followed by a value.

- `Result_File name`: the result will be stored in the `name` file instead of the default `.result_IA` file.
- `Iteration integer`: the maximum number of boxes that may be produced by the algorithm. Indicating this value is compulsory.
- `Debug integer`: the debug level (0=no debug, 1=low level debug, 2=full debug)
- `AccuracyF real`: the `epsilonf` parameter that will be used to determine a solution if Moore or Kantorovitch succeed.
- `Kanto real`: we will apply Kantorovitch test as soon as the maximal width of the input ranges is lower than real
- `Min_IA real, Mini_IA real`: if the change in a variable exceed a threshold ϵ we start again the analysis. We give here two values such that `Mini_IA` $\leq \epsilon \leq$ `Min_IA`. The default value is 0.001 for `Mini_IA` and 0.01 for `Min_IA`
- `Directory name`: if you have already processed the equations with MAPLE you may give here the name of the directory where the files resulting from the processing have been written. This avoid to re-process the equations.
- `Location_IS name`: the location of the library `lib_IS.m`
- `Level integer`: the level of bisection you want (from -1, no bisection to any positive number)
- `Use_Resultant`: the resultant of any two equations which are algebraic in any variable will be computed and will be used to discard input ranges. This option may add a significant amount of time to the pre-preprocessing. If you have a large number of equations you will end up with a large number of equations resulting from the computation of the resultant. You may however use only a subset of these equations by first processing the equations without this option, compute some resultant equations by hand and indicate that they are resultant equation by writing them in parser format in the file `/tmp/Resultant`, while indicating their number in the file `/tmp/DimensionResultant`. If you are using the option `-G` or `-H` of the analyzer you will have to provide also the gradient equations of the equations you are using in a file called `Gradient_Resultant`.
- `LevelC integer`: if the `integer` m is greater or equal to 0 the first test the analyzer will perform is to consider in turn each variable x_i and to compute the interval evaluation of the equations with x_i having as interval value first $[\underline{x_i}, \underline{x_i}+(\overline{x_i}-\underline{x_i})/(m+2)]$ and then $[\overline{x_i}-(\overline{x_i}-\underline{x_i})/(m+2), \overline{x_i}]$. If the interval evaluation of one equation has a constant sign, then the interval for x_i will be modified to $[\underline{x_i}+(\overline{x_i}-\underline{x_i})/(m+2), \overline{x_i}]$ in the first case and $[\underline{x_i}, \overline{x_i}-(\overline{x_i}-\underline{x_i})/(m+2)]$ in the second case. If one variable at least is modified with an amplitude of modification larger than `Mini_IA` the process is reiterated. Instead of using this process each time a variable is change we may use it only from time to time. This may be done by setting `LevelC` to a negative number lower or equal to -10. This number will indicate at the same time the number of variable changes that must occur before using the process and the value of m used for managing the interval. For example a value of -100 will indicate that the process will be used every 10 changes of variable (`rint(-LevelC)/10`) with $m = 0$ (m =10(`-LevelC`/10-`rint(-LevelC)`/10)). This process may be computer intensive and therefore the default value of `LevelC` is -1 so that it is not used.

- `UseFullCycle`: when using the bound theorems the analyzer will consider in turn each equation and then in turn each variable in which the current equation is algebraic. If an update in a variable has taken place with an amplitude larger than `Mini_IA` the analyzer will continue until all variables of the current equation has been considered and then will start again the analysis from equation 1. If you put this key-word in your configuration file the analyzer will start again the analysis only when all the equations have been processed. Turning on this option may be interesting if you have used the `LevelC` option as processing the equations is in general faster than using the combinatory approach: with this option you may have change on a larger number of unknowns which may lead to better result when using the combinatory approach.

For example

```
Iteration 1000
AccuracyF 0.00001
Debug 0
Kanto 0.5
Level 2
Location_IS /u/ALIAS/Maple
```

is a valid configuration file. It indicates that at most 1000 ranges may be created (this is not the maximal number of ranges at the end of the algorithm but the maximal number of ranges that may be created at any time in the algorithm), that the accuracy for the Newton scheme is 0.00001, that the algorithm will run quietly, that Kantorovitch will be used as soon as the maximal diameter of the considered range is lower than 0.5 and that the depth level will be 2. We indicate also in which directory is located the library `lib_IS`. files are located.

You use the configuration file with the syntax:

```
Interval_Analyzer -F [formula file] -G -H -R [range file] -C
[configuration file]
```

So a typical analysis and solving process will be done by using the two following command lines (which are basically identical to what is used in the script `script_solver` provided in the standard distribution of `ALIAS`):

```
Interval_Analyzer -F equation.maple -G -H -R range -C conf_analyzer
Interval_Solver -F /tmp/Equation -G /tmp/Gradient_Equation -H
/tmp/Hessian_Equation -R .result_IA -GI .gradient_IA -C conf_solver
```

You will then have the result of the solution of your system in the file `.result_IS`.

12.4.3 Dealing with inequalities

The generic analyzer may deal also with inequalities. Here we will add an argument to the generic analyzer whose syntax will be

```
Interval_Analyzer -F [formula] [-G -H] -R [range] -I [inequalities]
```

`inequalities` is a MAPLE file, called the *inequalities file* that describe equations in the same manner than a formula file. But the analyzer will output as possible ranges only the ranges for which all the equations in the inequalities file may be positive or equal to 0 and eliminate the ranges for which at least one equation is strictly negative.

12.4.4 Pre-processing and dealing with parametric equations

12.4.4.1 Pre-processing

In some cases it may be interesting to analyze a given system of equations with various initial input ranges. In the current state the system will be processed again and again for each input ranges. If this system is large this result in unnecessary computation time. When processing for the first time a system the generic analyzer will produce intermediary files in parser format in the `/tmp` directory. If you use the `-G` and `-H` option of the analyzer the intermediary files will be:

- `Equation`: the system of equation
- `Gradient_Equation`: the gradient equations of the system
- `Hessian_Equation`: the hessian equations of the system
- `DimensionEq`: the *dimension file* i.e. a file which contains a list of integer which indicates the degree of the first equation in the first unknown, in the second unknowns, ..., then a similar list for the second equation, and so on
- `CoeffEq`n-m: the coefficients of the equation n of the system in the unknown number m
- `Gradient_CoeffEq`n-m: the gradient of the coefficients of the equation n of the system in the unknown number m

Note also that you may have to be careful when stopping the analyzer when it is generating these files. Indeed on some systems these files may be read-write protected even for the owner. If you run again the analyzer MAPLE will not be able to write the same files and will produce an error message: you will then need to remove by hand each of the created files and start again.

As soon as these files have been generated once it is not necessary to re-compute them. To indicate that these quantities have been already processed it is is sufficient to include the following sentence in the configuration file:

```
Directory name
```

Instead of computing these files the analyzer will look for them in the directory `name`. Thus for example after running the analyzer for the first time without this sentence in the configuration file you may analyze the same system without processing the file by indicating in your configuration file:

```
Directory /tmp
```

12.4.4.2 Parametric equations

It may also be interesting to use the pre-processing for parametric systems, i.e. for systems for which some coefficients will be described in analytical form in the MAPLE formula file, but which are not unknowns. An additional argument for the generic analyzer will be a file, called the *parameter file* which will give specific values for the parameter. For example in the equation:

```
eq1:=x^2-x*cos(y)-x1:
```

`x1` may be considered as a parameter. On the first run of the analyzer the intermediary files described in the previous section will be generated and at run time the value of `x1` will be substituted by the value indicated in the parameter file. After this first run and if you want to analyze the system with another value for `x1` you may use the pre-processing mechanism described in the previous section and just change the value of `x1` in the parameter file. See section 12.4.6.1 for an example of the analysis of parametric system.

Each line of the parameter file indicates a name of a parameter and its value. Thus

```
x1 1
```

is a valid parameter file. To indicate a parameter file to the analyzer use the `-P` argument followed by the name of the parameter file.

A special case of parameter may be used to speed up the computation. Assume that you have a system with large numerical constants (by "large" we mean that the string representing the numerical values has a large number of characters). Each evaluation of your system will require first to read the strings representing the numerical constants and then to convert it in a float number. It may therefore be interesting to store once the constants values and to substitute them in the analytical form of the equations by symbols which may be attached to these values. This may be done by substituting the numerical constants by symbols of the form `_IA`n where n is an integer starting from 0. These parameters have then to be stored in a parameter file. Thus, for example, consider the equations:

```
eq1:=1.12345678901*x1+4.563213456789222:
```

Using MAPLE you may easily change this equation to:

```
eq1:=_IA0*x1+_IA1:
```

and store the values in a parameter file:

```
_IA0 1.12345678901
_IA1 4.563213456789222
```

When reading the parameter file the program will store in a special array the value of the parameters whose name start by the symbol `_IA`. Then, when the parser find a symbol starting with this symbol it will determine the number following this keyword (which is the value of the index in the parameter table) and will just substitute the corresponding value found in the parameter table. Hence when evaluating an expression the parser will have to read less character and avoid unnecessary conversion from a string to a double. Note that these parameters must be numbered in sequence, starting from 0.

12.4.5 Errors and Debug

The analyzer may generate the following error code:

- -1: too many ranges have been generated during the algorithm. Note that this test is only approximate.
- -2: there is an error in the files generated for the parser
- -4: the number of equations in your system is not correct (this should not occur with the generic analyzer but may happen if you use it in a program)
- -11: the parameters `equation_processed` of the analyzer is not 0 or 1 (cannot occur for the generic analyzer)
- -20: the dimension file cannot be opened
- -40: the degree of the equations in the various unknowns is 0 (should not occur for the generic analyzer)

You may get some debug information by setting the variable `Debug_Level_Solve_General_Interval` to 1 (minimal level of debug information) or 2 (full debug).

12.4.6 Examples

12.4.6.1 Non algebraic equations

Let consider the following system

```
eq1:=x^2-x*cos(y)-1:
eq2:=y^2-cos(x)-1:
```

for which we want to determine if real roots exist in the range [-5,5], [-5,5]. With a depth level 0 (no bisection is done) the analyzer proposes the following ranges as possibly containing a real root:

```
Range 1( x y)
[-0.86812856880248989722,-0.86812856880248989722]
[-1.28306500467802298,-1.28306500467802298]
Range 2( x y)
[1.219833908062562422,1.219833908062562422]
[-1.1592247392497665448,-1.1592247392497665448]
Range 3( x y)
[1.2198350997243703198,1.2198367316341245381]
[-1.159223547587958647,-1.1592195879408853099]
Range 4( x y)
[-0.86812856880248978619,-0.86812856880248978619]
[1.28306500467802298,1.28306500467802298]
Range 5( x y)
[1.219833908062562422,1.219833908062562422]
[1.1592247392497667668,1.1592247392497667668]
Range 6( x y)
[1.2198352388536839452,1.2198367316341249822]
[1.1592195879408850878,1.1592236867169092296]
```

Among this 6 ranges, 4 are reduced to point and are therefore a result of the application of the Newton scheme. The other 4 ranges are very close to one of the solution, but the algorithm has not been able to eliminate them. With a depth level of 1 the algorithm proposes directly the four solutions of this system:

```
Range 1( x y)
[-0.86812741994331155126,-0.86812741994331155126]
[-1.2830653469203250339,-1.2830653469203250339]
Range 2( x y)
[1.2198339926936523359,1.2198339926936523359]
[-1.1592245852349318813,-1.1592245852349318813]
Range 3( x y)
[-0.86812741994331166229,-0.86812741994331166229]
[1.2830653469203250339,1.2830653469203250339]
Range 4( x y)
[1.219833901899016082,1.219833901899016082]
[1.159224777165462017,1.159224777165462017]
```

Note that the initial range [-5,5] may be largely expanded without modifying the result and with only a low amount of additional computation time. For example for the range [-1000,1000] for both variables the computation time remains the same.

The previous system may be considered as a special occurrence of the system:

```
eq1:=x^2-x*cos(y)-x1:
eq2:=y^2-cos(x)-1:
```

where the parameter `x1` has value 1. This special case of the generic system may be analyzed by indicating in a parameter file:

```
x1 1
```

Now just by changing the value of `x1` in this file to

```
x1 2
```

and modifying the configuration file to include the sentence:

```
Directory /tmp
```

we will get the solutions:

```
Range 1( x y)
[-1.2271026453420417202,-1.2271026453420417202]
[-1.1562745250557344701,-1.1562745250557344701]
Range 2( x y)
[1.757647484313144215,1.757647484313144215]
[-0.90234927430355182931,-0.90234927430355182931]
Range 3( x y)
[-1.2271026453420417202,-1.2271026453420417202]
[1.1562722309470763182,1.1562722309470763182]
Range 4( x y)
[1.757647484313144215,1.757647484313144215]
[0.9023492743035520513б,0.90234927430355205136]
```

12.4.6.2 Using the generic analyzer in a program

You may use directly the analyzer in a program using the following syntax:

```
int Equation_Analyzer(int Dimension,int Dimension_Eq,int MaxBox,
         char *formula_file,char *gradient_file,char *hessian_file,
         char *inequalities_file,char *dimension_file,char *coeff_file,
         char *gradient_coeff_file,
                      int method,int nb_inequalities,int equation_processed,
                      INTERVAL_VECTOR &Input,
                      double Acc_Var,double Acc_Eq,double Acc_Kanto,
                      int Depth_Level,
                      INTERVAL_MATRIX &Range,
                      int *Nb_Range);
```

with:

- `Dimension`: the number of unknowns in the system
- `Dimension_Eq`: the number of equations in the system
- `MaxBox`: the maximum number of ranges that may be produced at any step of the algorithm
- `formula_file`: if the system has already been processed the name of the file which contain the equations in parser format
- `gradient_file`: if the system has already been processed the name of the file which contain the gradient equations in parser format
- `hessian_file`: if the system has already been processed the name of the file which contain the gradient equations in parser format
- `inequalities_file`: if the system has already been processed the name of the file which contain the inequalities equations in parser format. The number of inequalities must be given in `nb_inequalities`
- `dimension_file`: the name of the dimension file, i.e. the file which contain the degree of the equations in the unknowns (see section 12.4.4.1). If this string has 0 length the analyzer will assume that you have filed the integer array `Dimen_IA(Dimension_Eq,Dimension` with the appropriate number: for example `Dimen_IA(1,2)` should contain the degree of the first equation in the second variable (the order of the variable is their order of occurrence in the string array that contain their names).
- `coeff_file`: the base-name for the files containing the coefficients of the equations. For example `coeff_file1-2` must contain the coefficients of the first equation considered as a univariate polynomial in the second variable
- `gradient_coeff_file`: the base-name for the files containing the gradient of the coefficients of the equations. For example `gradient_coeff_file1-2` must contain the gradient of the coefficients of the first equation considered as a univariate polynomial in the second variable
- `method`: an integer describing the method you are using for evaluating the equation:
  - 0: straightforward interval evaluation
  - 1: evaluation using the gradient of the equation
  - 2: use Kantorovitch and Newton scheme to determine roots of the system and intervals containing unique roots of the system.
- `nb_inequalities`: the number of inequalities constraints you have
- `equation_processed`: an integer which may be 0 or 1.
  - 0: you have not processed your system. The analyzer will consider that the argument `formula_file` is a MAPLE program that contain the description of your equations. It will then create all the necessary intermediate files (see section 12.4.4.1). Note that in that case all the string files of this procedure will be substituted by the names of the corresponding intermediary files. The storage space of these strings must therefore be sufficient.
  - 1: the system has already been processed and the name indicated in the arguments of this procedure are parser files
- `Input`: the initial set of ranges for the unknowns
- `Acc_Var`: unused at this time
- `Acc_Eq`: if `method` is 2 the threshold under which an equation will be considered to be equal to 0 in the Newton scheme

- `Acc_Kanto`: if `method` is 2, the maximal diameter in a set of ranges under which we start applying Kantorovitch. A typical value is 0.5
- `Depth_Level`: the level of bisection that will be used by the analyzer (from 0 to some positive value)
- `Range`: the sets of ranges produced by the algorithm
- `Nb_Range`: the number of sets in `Range`

Beside these arguments you must fill the string array `variable_name_IS` with the name of the unknowns. If you have a parametric system you must indicate the name of the parameters in the string array `parameter_name_IS`, their values in the array of doubles `parameter_value_IS` and set the integer variable `Nb_Parameter_IA` to the number of parameters.

12.5 Using the parser programs

The parser programs are available in the library `lib_Parser.a`. You will have to include the following header files:

```
header_parser.h
header_Solve_General_Interval.h
header_Utilities_Interval.h
```

If you are planing to use the analyzer you must include the library `lib_UP.a`. In your program you must avoid to use variables whose name include the symbols `_IA` and `_IS`.

The following programs are equivalent to the standard `ALIAS` programs except that they use the parser.

```
int Kantorovitch_Analyzer(int Dimension,
                 char *Function_File,char *Gradient_File,
                 char *Hessian_File,
                 VECTOR &Input,double *eps);

int Newton(int Dimension,
          char * TheIntervalFunction_File,
          char * Gradient_File,
          VECTOR &Input,double Accuracy,int Max_Iter,VECTOR &Residu);

int Solve_General_Interval(int Dimension_Var,int Dimension_Eq,
        char * TheIntervalFunction_File,
                INTERVAL_VECTOR & TheDomain,
                int Order,
                int Iteration,
                int Stop_First_Sol,
                double Accuracy_Variable,
                double Accuracy,
                double Dist_Sol_Diff,
                INTERVAL_MATRIX & Solution,int Nb_Max_Solution,
                int nb_inequalities,char *Ineq_File);
int Solve_General_Gradient_Interval(int Dimension,int Dimension_Eq,
        char * TheIntervalFunction_File,
                char * Gradient_File,
                INTERVAL_VECTOR & TheDomain,
                int Order,
                int Iteration,
                int Stop_First_Sol,
                double Accuracy_Variable,
```

```
                double Accuracy,
                double Dist_Sol_Diff,
                INTERVAL_MATRIX & Solution,int Nb_Max_Solution,
                int nb_inequalities,char *Ineq_File);
int Solve_General_Gradient_Interval(int Dimension,int Dimension_Eq,
        char * TheIntervalFunction_File,
                char * Gradient_File,
                INTERVAL_VECTOR & TheDomain,
                int Order,
                int Iteration,
                int Stop_First_Sol,
                double Accuracy_Variable,
                double Accuracy,
                double Dist_Sol_Diff,
                INTERVAL_MATRIX & Solution,int Nb_Max_Solution,
                INTEGER_VECTOR &Init_Grad,
                int nb_inequalities,
                char *Ineq_File);
int Solve_General_JH_Interval(int Dimension,int Dimension_Eq,
        char * TheIntervalFunction_File,
                char * Gradient_File,
                char * Hessian_File,
                INTERVAL_VECTOR & TheDomain,
                int Order,
                int Iteration,
                int Stop_First_Sol,
                double Accuracy_Variable,
                double Accuracy,
                INTERVAL_MATRIX & Solution,
                INTEGER_VECTOR & Is_Kanto,
                int Apply_Kanto,
                int Nb_Max_Solution,
                int nb,char *file);
```

Chapter 13

Parallel processing

13.1 Parallelizing `ALIAS` programs

Interval analysis has a structure that is appropriate for a distributed implementation. Indeed the principle of the algorithm is to process a list of boxes, the processing for a box being independent of the other boxes in the list (except that it may have be stopped). A classical distributed implementation will rely on the master/slave paradigm. A master computer will manage the list of boxes and monitor the activity of the slaves. As soon as a slave is free the master will send a box to this slave. The slave, which has its own list of boxes, will perform a few iteration of the solving algorithm and then send back to the master the remaining boxes to be processed in its list and eventually the solutions it has found. After receiving a signal from the slave the master will receive the boxes from the slave and append them to its list of boxes. This slave being now free the master will send him another box in its list.

Hence most of the calculation will be performed by the slaves. It may happen however that at a given time all the slaves are busy processing boxes. The master itself may then start a few iteration of the solving algorithm on the current box of its list while carefully monitoring the activity of the slaves.

The structure of the interval analysis algorithms is very convenient for using the above paradigm:

- few data are to be transmitted by the master to the slave: typically for a problem with n unknowns the master will have to send $2n$ floating numbers that describe a box. It may however be interesting to send other information such as the derivatives but still the amount of data will be low
- the processing time of a box is in general much more large than the transmission time

The gain in computation time will in general be lower than the number of used slaves. This is due to

- the transmission time between the master and the slaves
- an overhead of the slave program: in our implementation the slaves uses basically the same program than the master and perform some coherence checks on the unknowns, equations, available memory size ... before starting the processing of the box
- the fact that only one computer manages the list of boxes. This master may be still managing the boxes send by a slave while others slaves have already finished their processing. Clearly the master/slave paradigm exposed above may not be the best for a large number of computers. In a next version we intend to propose another paradigm for the use of the interval analysis algorithms in the framework of *grid computing*

Note however that in some cases the gain may be larger than the number of slaves. This is, for example, the case of the optimization algorithms as the independence of box processing is no more true: the discovery of a good optimum by a slave may leads to stop the processing of the other slaves.

In any case a key point of a distributed implementation is to be able to

- stop an algorithm that is run by a slave computer after it has performed some processing on a box
- recover the current state of the slave process i.e. the solutions if any and the remaining unprocessed boxes.

13.2 Stopping a procedure and miscellaneous utilities

There are different methods to stop the current bisection process:

1. the number of boxes still to be processed is greater than a given threshold N: this is obtained by setting the global variable `ALIAS_Parallel_Max_Bisection` to $N-1$. . A safety procedure may be used as the number of bisection may be large before getting the right number of boxes (for example when using the single bisection mode): the maximal number of bisection may be indicated by setting the global variable `ALIAS_Parallel_Max_Split` and the procedure 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).

2. the number of performed bisection is greater than a given threshold M: this is obtained by setting the global variable `ALIAS_Parallel_Max_Bisection` to $-M$

3. 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 variable `ALIAS_Parallel_Max_Bisection` to $-M$ and the global 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 the procedure will return the error code -1. The value of S is given by the global variable `ALIAS_Safety_Factor` with a default value of 2

4. 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 variable `ALIAS_Parallel_Max_Bisection` to N and the global variable `ALIAS_Parallel_Max_Reverse` to M

5. 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 double `ALIAS_TimeOut` which indicates the maximum amount of time (in minutes) during which a slave may run (this amount will be respected only approximatively).

The number of the box still to be processed may be determined from the integer array `ALIAS_Parallel_Box`. Let `ALIAS_Parallel_Box[0]`$=i$ and `ALIAS_Parallel_Box[1]`$=j$. If $i > j$, then no boxes remains to be processed otherwise the box numbers to be processed start at i and finish at j (hence there are $j-i+1$ boxes to be processed). The boxes are stored in the interval matrix array `Box_Solve_General_Interval`. If the system has n unknowns, the box for each unknown in the box numbered i are given by `Box_Solve_General_Interval(i,1)` ... `Box_Solve_General_Interval(i,n)`. For procedures involving the gradient of the functions you may retrieve the simplified gradient in the integer matrix `Gradient_Solve_General_Interval` while for the procedures involving the gradient and hessian of the functions the gradient may be retrieved in the interval matrix `Gradient_Solve_JH_Interval`. Note that this storage may not be available if you have set the flag `ALIAS_Store_Gradient` to 0 (its default value is 1).

The variable `ALIAS_Parallel_Slave` should be set to 0 for a sequential use of the algorithm, to 1 if the algorithm is run by a master computer and to 2 if the algorithm is run by a slave computer.

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

If the master/slave scheme is used the master may perform some steps of a solving algorithm but for efficiency purpose it is necessary that the master monitor the slaves to determine if one has completed its calculation. This is the role of the `ALIAS_Slave_Returned` procedure that stops the processing done by the master as soon as a slave is free. Such a procedure may be found in the `lib_Slave.a` library: it is based on the `pvm` message passing mechanism (see the ALIAS-Maple manual). For the non parallel processing the equivalent procedure may be found in the `lib_No_Slave.a` library. Hence according to a sequential or a parallel use one of this library has to be linked to the main program.

For solving procedures involving the use of the gradient we may use two schemes:

- the master transmits to the slave both the boxes and the simplified gradient

- the master transmits to the slave only the boxes and the slave will first compute the simplified gradient before running the solving algorithm

In the first case the transmission time of the gradient may be relatively large and the master may spend a large amount of time for this transmission while other slaves are currently free. Hence it may be interesting to have a flag which indicate which mode is used: this is the role of the integer `ALIAS_Transmit_Gradient`.

The ALIAS-Maple library offers a parallel implementation of some solving algorithm. Communication between the slaves and the master is ensured by the message passing mechanism `pvm`. See the ALIAS-Maple manual for a detailed account of the distributed implementation.

This implementation uses some specific C++ procedures.

```
void ALIAS_Prepare_Buffer(int DimVar,INTERVAL_VECTOR &X,char *buffer)
```

which put in the string `buffer` the lower and upper bounds of the `DimVar` ranges contained in the interval vector `X`.

The master and slaves exchange strings and in the parallel implementation offered currently by `ALIAS` all data are contained in these strings (this may change in the near future as `PVM` offers other possibility). Each numerical data is introduced by few control character. The following control character are used:

- `B`: the next floating point numbers will be a box that is sent back by the slave
- `BS`: same as `B` except that the box may be used as a parameter by the slave (for example the location of the current optimum)
- `F`: is followed by an integer. It's used by the slave to indicate how many solutions it will send to the master
- `N`: just used to indicate that a slave has completed its task and has nothing to send back to a slave
- `P`: followed by the control character `I` or `F`. Its allow to transmit to the slave an integer or a float that will be used as a parameter
- `S`: the next floating point numbers will be a solution box that is sent back by the slave
- `SP`: followed by an integer. It's used by the master to indicate to the slave the value of the parameter `ALIAS_Parallel_Max_Bisection`

Using this convention we may used the following procedure to decode the string sent by the master or the slave:

```
int ALIAS_Read_Buffer(char *mess,
          int Nb_Var, int Nb_Eq,INTERVAL_MATRIX &Box,int *Nb_Total,
          INTERVAL_MATRIX &Sol,int *Nb_Sol,int *Nb_Split,
          INTEGER_VECTOR &IPar,int *Nb_Ipar,
          VECTOR &FPar,int *Nb_Fpar,INTERVAL_MATRIX &BoxPar,
          int *Nb_BoxPar)
```

where `mess` is the string and

- `Box`: the `Nb_Total` boxes with the control character `B`
- `Sol`: the `Nb_Sol` boxes with the control character `S`
- `Nb_Split`: the integer following `F`
- `IPar`: the `Nb_Ipar` integer following the control character `P I`
- `FPar`: the `Nb_Fpar` float following the control character `P F`
- `BoxPar`: the `Nb_BoxPar` parameter boxes following the control character `BS`

If is necessary we may transmit also the simplified jacobian associated to the boxes using the procedure:

```
int ALIAS_Read_Buffer_Gradient(char *mess,
          int Nb_Var, int Nb_Eq,INTERVAL_MATRIX &Box,
          INTEGER_MATRIX &GBox,int *Nb_Total,
          INTERVAL_MATRIX &Sol,int *Nb_Sol,int *Nb_Split,
          INTEGER_VECTOR &IPar,int *Nb_Ipar,
          VECTOR &FPar,int *Nb_Fpar,INTERVAL_MATRIX &BoxPar,
          int *Nb_BoxPar)
```

Here the boxes followed by the simplified jacobian `GBox` are introduced by the control character `BG`.

If the true jacobian has to be transmitted we use:

```
int ALIAS_Read_Buffer_Hessian(char *mess,
          int Nb_Var, int Nb_Eq,INTERVAL_MATRIX &Box,
          INTERVAL_MATRIX &GBox,int *Nb_Total,
          INTERVAL_MATRIX &Sol,INTEGER_VECTOR &Is_Kanto,
          int *Nb_Sol,int *Nb_Split,
          INTEGER_VECTOR &IPar,int *Nb_Ipar,
          VECTOR &FPar,int *Nb_Fpar,INTERVAL_MATRIX &BoxPar,
          int *Nb_BoxPar)
```

Here the boxes followed by the jacobian `GBox` are introduced by the control character `BG`. Furthermore each solution is followed by an integer that indicates the type of the solution and can be accessed by `Is_Kanto`.

Chapter 14

How to install and use ALIAS

14.1 Installing ALIAS

ALIAS is available through anonymous ftp at ftp-sop.inria.fr, directory coprin/logiciels/ALIAS.

The standard distribution of ALIAS is contained in different directories:

- Doc: documentation (manual[version-number].ps)
- Examples: test programs using ALIAS
- Lib-linux, Lib-solaris: the C++ library for linux and solaris. including the necessary header files
- Profil, Profil-linux: the BIAS/Profil library composed of .a library and header files located in the incl sub-directory
- Maple: the directory which contain the ALIAS-Maple library It is composed of the library ALIAS.m and a directory ALIAS that contain all the necessary modules. An on-line help is available. This directory also include the lib_IS.m library that is used by the Analyzer.

The library directory includes the following header files:

```
header_2B.h
header_Analysis_Univariate.h
header_Solve_General_Interval.h
header_Solve_UP.h
header_Trigo.h
header_Utilities_Interval.h
header_Utilities_UP.h
header_constants.h
header_externe.h
header_functions.h
header_parser.h
```

and the following library files:

```
lib_2B.a
lib_Continuation.a
lib_No_Slave.a
lib_Slave.a
lib_Parser.a
lib_Solve_General.a
lib_Solve_General_Parser.a
lib_Solve_Simplex.a
lib_Trigo.a
lib_UP.a
```

The library `lib_2B.a` includes the program that are necessary to run the 2B filtering (see section 2.17). The library `lib_Continuation.a` contains the programs described in section 9.1, the library `lib_Solve_Simplex.a` the programs devoted to the specific system solving procedures described in sections 2.14, 2.16. The general purpose solving algorithms and various utilities are installed in the library `lib_Solve_General.a` which must be included in the last position in the makefile.

The library `lib_UP.a` includes all the programs related to univariate and parametric polynomial, `lib_Trigo.a` contains some utilities for trigonometric equations and the libraries for using the parser are `lib_Parser.a` and `lib_Solve_General_Parser.a`.

The libraries `lib_Slave.a`, `lib_No_Slave.a` are to be used respectively if a parallel or sequential implementation of ALIAS is used.

To install the library you must first install the `Profil` library in a specific place e.g. `/u/Profil` which will contain the `.a` files of `Profil` together with its header files in the sub-directory `incl`. *Note that this distribution of BIAS/Profil is somewhat different from the standard distribution as some bugs have been corrected.*

You will then install the `ALIAS` library in a specific directory, e.g. `/u/ALIAS`, which will contain the above library and header files.

The standard distribution includes also three programs based on the `ALIAS` parser:

- `Evaluation`: a simple program to get the interval evaluation of formula being given range on the unknowns
- `Interval_Analyzer`: a program which produces sets of ranges that may contain the real roots of a system (see section 12.4. These sets are an improvement of an initial set of range
- `Interval_Solver`: a program that take as input the result of `Interval_Analyzer` and compute the roots of the system (see section 12.2)
- `lib_IS.m`: a MAPLE library which contain tools for writing equations in a format that can be used by the parser. The location of this library must be indicated in the configuration file of the Analyzer.
- `script_solver`: a Unix script that take as argument the name of a formula file written in MAPLE format and a range file. With this script the analysis and solving of the system is performed in sequence. This script uses two configuration files, `conf_analyzer` and `conf_solver`, which are used by the analyzer and the solver. Before using this script you must update the location of the `lib_IS.m` library in the configuration file `conf_analyzer` by changing the directory name following the key-word `Location_IS`. This script has some optional arguments:
  - `-I ineq`: indicate that the system has inequalities defined in the MAPLE file `ineq`
  - `-CA confa`: indicate that the analyzer must use the configuration file `confa` instead of the default one
  - `-CS confs`: indicate that the solver must use the configuration file `confs` instead of the default one

14.2 Header and linking

In most cases the following include files should be provided at the beginning of the program:

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

For using specific procedures of this package it is necessary to include the following header file:

```
header_constants.h
```

For the solving algorithm, optimization and continuation algorithms it is necessary to include:

```
header_Solve_General_Interval.h
header_Utilities_Interval.h
```

and use the libraries `lib_Solve_General.a`, `lib_Solve_Simplex.a` and `lib_Continuation.a`.

For analyzing and solving univariate polynomial you need:

```
header_externe.h
header_Analysis_Univariate.h
header_Utilities_UP.h
header_Solve_UP.h
```

and use the library `lib_UP.a`. If you intend to solve and analyze trigonometric equation you need:

```
header_externe.h
header_Analysis_Univariate.h
header_Utilities_UP.h
header_Trigo.h
```

and use the library `lib_UP.a` and `lib_Trigo.a`. For using the parser you need:

```
header_parser.h
```

and the libraries `lib_Solve_General_Parser.a`, `lib_Parser.a`.

In addition to these libraries you have to add either `lib_No_Slave.a` or `lib_Slave.a` respectively for a sequential and parallel implementation.

The link flags for using BIAS/Profil are:

```
-lProfilOpt -lProfilNLI -lProfilMISC -lProfil -lBias
```

with the following flags (if you have installed the `Profil` library in `/u/Profil` and the `ALIAS` library in `/u/ALIAS`):

```
-I/u/Profil/incl -L/u/Profil -L/u/ALIAS
```

The compilation flags must include the `-fpermissive` options that is necessary for using `Profil`. Hence a typical makefile will look like:

```
 .C.o:
g++ -O -fpermissive -I$(INCL) -w -c -o $*.o $<
CC=g++ -g -w -O -fpermissive
INCL=/u/ALIAS/Profil/incl
LIB=/u/ALIAS/Profil
INCLI=/u/ALIAS/Lib-solaris
LIB_SOLVE=${INCLI}/lib_Solve_General.a ${INCLI}/lib_No_Slave.a
LIB_UP=${INCLI}/lib_UP.a
LIB_TRIGO=${INCLI}/lib_Trigo.a
LIB_SIMPLEX=${INCLI}/lib_Solve_Simplex.a
LIB_CONTINUATION=${INCLI}/lib_Continuation.a
LD= -lProfilOpt -lProfilNLI -lProfilMISC -lProfil -lBias

Test:Test.o
$(CC) -I$(INCL) Test.C  -o Test \
-L$(LIB) $(LIB_UP) $(LIB_TRIGO) $(LIB_CONTINUATION) \
$(LIB_SIMPLEX) $(LIB_SOLVE) $(LD) -lm
```

Note that for using the current version you still need to use the Gnu C++ compiler version 2.95 and not higher. Indeed the BIAS/Profil package cannot be compiled with a higher version as it uses an "old" version of C++ (whatever this means for a language that is intended to be reusable ...).

If you plan to use `pvm` for making a parallel implementation of your program then the flag `-lm` should be substituted by `-lm -lpvm3 -lnsl -lm -ldl -lintl -lsocket` on a SUN workstation and by `-lm -lpvm3 -lnsl -lm -ldl` on a PC running Linux. The `LIB_SOLVE` dependency has also to be modified to:

```
LIB_SOLVE=${INCLI}/lib_Solve_General.a ${INCLI}/lib_Slave.a
```

for a better efficiency.

14.3 Reserved variables

The names and default values of the global reserved variables are, by type:

`int ALIAS_Algo_Find_Optimum=0`	`int ALIAS_Algo_Has_Optimum=0`
`int ALIAS_Allow_Backtrack=1`	`int ALIAS_Allow_Storage=0`
`int ALIAS_Full3B=0`	`int ALIAS_Func_Has_Interval=0`
`int ALIAS_Guide_Bisection=0`	`int ALIAS_Has_Optimum=0`
`int ALIAS_Mixed_Bisection=0`	`int ALIAS_ND=0`
`int ALIAS_Opt_Func`	`int ALIAS_Optimize=0`
`int ALIAS_Order`	`int ALIAS_Parallel_Box[2]`
`int ALIAS_Parallel_Max_Bisection=0`	`int ALIAS_Parallel_Max_Box=-1`
`int ALIAS_Parallel_Max_Reverse=30`	`int ALIAS_Parallel_Max_Split=0`
`int ALIAS_Parallel_Slave=0`	`int ALIAS_Problem_Continuation=0`
`int ALIAS_Pure_Equation=0`	`int ALIAS_Round_Robin=1`
`int ALIAS_Safety_Factor`	`int ALIAS_Safety_Factor=2`
`int ALIAS_Stop=0`	`int ALIAS_Stop_Sign_Extremum=0`
`int ALIAS_Store_Gradient=1`	
`int ALIAS_Use3B=0`	`int ALIAS_Use_Kraw_Continuation=0`
`int ALIAS_Use_SubEq3B=0`	`int Debug_Level_Solve_General_Interval=0`
`int Enable_Delete_Fast_Interval=1`	`int Interval_Combinatory_Level=-1`
	`int Interval_Evaluate_Equation_Alone=0`
`int Max_Iter_Laguerre_Interval`	`int Max_Iter_Newton_JH_Interval`
`int Reverse_Storage`	`int Reverse_Storage=0`
`int Single_Bisection=0`	`int Switch_Reverse_Storage=-1`

`double ALIAS_Accuracy_Extremum=1.e-6`	`double ALIAS_Bound_Smear=1.e-5`
`double ALIAS_Extremum`	`double ALIAS_Extremum_Max`
`double ALIAS_LO_A=0.9`	`double ALIAS_LO_B=1.1`
`double ALIAS_Seuil_Ajuste=0.5`	`double ALIAS_Switch_3B=0`
`double ALIAS_Threshold_Optimiser=100`	`double Min_Step_Laguerre_Interval`
`double Step_Laguerre_Interval`	

`INTERVAL ALIAS_Algo_Optimum`	`INTERVAL ALIAS_Algo_Optimum_Max`
`INTERVAL ALIAS_Optimum`	`INTERVAL ALIAS_Optimum_Max`

`INTERVAL_MATRIX ALIAS_Vector_Optimum`	`INTERVAL_MATRIX Box_Solve_General_Interval`
`VECTOR Coeff_Solve_JH`	`INTERVAL_VECTOR Coeff_Safe_Solve_JH`
`INTEGER_MATRIX Gradient_Solve_General_Interval`	`INTERVAL_MATRIX Gradient_Solve_JH_Interval`
`INTERVAL_VECTOR Interval_Solution_UP`	`INTERVAL_VECTOR Local_Sol_JH`

You must also avoid to use variable names starting with `ALIAS_`. If you are using the parser you should avoid having symbols name ending with `_IA` or `_IS`.

14.4 Changes with the previous version

A ⋆ indicates a change which is valid both in the ALIAS-C++ library and in the ALIAS-Maple library.

- ⋆ introduction of the Rouche theorem that may determine a ball with a single root of a square system of equations

- $\star$ `IntervalNewton`, a sophisticated interval Newton method for system of equations
- implementation of the `Miranda` theorem for verifying the presence of a solution in a box
- $\star$ systematic use of the Newton scheme in a solving scheme with the procedure `TryNewton`
- $\star$ the procedure `Divide_Evaluate_Multiple` allows to write an univariate polynomial P under the form $P(x) = (x - a_1) \dots (x - a_n)Q(x) + R$ and to interval evaluate the polynomial for some range for x
- `Divide_Polynom` allows one to divide an univariate polynomial with interval coefficients by another polynomial with interval coefficients
- generalized inverse trigonometric functions with `Filtre_Arc_Sin` and `Filtre_Arc_Cos`
- procedure for computing the inverse of an interval matrix
- procedure for computing the inverse of an interval matrix
- procedure for computing the centered form of expressions (`Centered_Form`)
- $\star$ procedure for computing the bicentered form of expressions (`BiCentered_Form`)
- procedure for calculating the binomial
- procedure for calculating the coefficient of the expansion of $(x - a)^n$
- procedure for calculating the centered form of a polynomial (`Derive_Polynomial_Expansion`)
- procedure for computing the coefficients of the unitary polynomial associated to a given polynomial (`Unit_Polynom`)
- implementation of the Weyl filter for univariate and parametric polynomials

Chapter 15

Examples of application of ALIAS-C++

15.1 Examples presented in this documentation

We describe here the examples used in the documentation. When installing ALIAS you will find a directory Examples providing other examples. Other examples of the use of interval analysis may be found on the COPRIN web page.

15.1.1 Example 2

This example comes from a planar robotics problem. Let the end-effector of the robot be a triangle. Each vertex of this triangle is connected to the ground via a linear actuator whose extremity is fixed point. We know the lengths of the three linear actuators (50, 26, 25) and we want to determine the pose of the end-effector i.e. the location of the vertex B_1 and its orientation. We have therefore to solve a system of three equations in three unknowns.

The following Maple program enable to get the equations of this example.

```
with(linalg):
#location of the linear actuator on the ground
xa1:=0:ya1:=0:xa2:=10:ya2:=0:xa3:=3:ya3:=10:
#location of the linear actuator on the end-effector
xb1:=0;yb1:=0:xb2:=4:yb2:=0:xb3:=2:yb3:=2:
for i from 1 to 3 do
OA.i:=array([xa.i,ya.i]):
CBr.i:=array([xb.i,yb.i])
od:

rot:=array([[cos(teta),-sin(teta)],[sin(teta),cos(teta)]]):
for i from 1 to 3 do CB.i:=multiply(rot,CBr.i): od:
OC:=array([x,y]):
for i from 1 to 3 do
      AB.i:=evalm(OC-OA.i):
      AB.i:=evalm(AB.i+CB.i):
      ro.i:=dotprod(AB.i,AB.i,'orthogonal'):
      ro.i:=simplify(ro.i):
od:

eq1:=ro1-50:eq2:=ro2-26:eq3:=ro3-25:
```

The system admit the two solutions:

$$(5,5,0) \qquad (3.369707132, 6.216516219, -0.8067834380)$$

Here we may use the general solving algorithm (section 2.3), the general solving algorithm with the gradient (section 2.4), the general solving algorithm with the gradient and Hessian (section 2.5).

These algorithms are implemented in the test program `Test_Solve_General1`, `Test_Solve_Gradient_General1`, `Test_Solve_JH_General1`

15.1.2 Example 3

In this example we will solve a trigonometric equation. This equation is derived from the previous example by subtracting the first equation to the second and third: this lead to a linear system in the unknowns `x,y` which are substituted in the first equation.

The following `Maple` program enable to get the equations of this example.

```
with(linalg):
xa1:=0:ya1:=0:xa2:=10:ya2:=0:xa3:=3:ya3:=10:
xb1:=0;yb1:=0:xb2:=4:yb2:=0:xb3:=2:yb3:=2:
for i from 1 to 3 do
      OA.i:=array([xa.i,ya.i]):
      CBr.i:=array([xb.i,yb.i])
od:

rot:=array([[cos(teta),-sin(teta)],[sin(teta),cos(teta)]]):
for i from 1 to 3 do CB.i:=multiply(rot,CBr.i): od:
OC:=array([x,y]):
for i from 1 to 3 do
      AB.i:=evalm(OC-OA.i):
      AB.i:=evalm(AB.i+CB.i):
      ro.i:=dotprod(AB.i,AB.i,'orthogonal'):
      ro.i:=simplify(ro.i):
od:

eq1:=ro1-50:eq2:=ro2-26:eq3:=ro3-25:

en1:=eq2-eq1:en2:=eq3-eq1:
sw:=solve({en1,en2},{x,y}):
assign(sw):
eq1:=numer(eq1):
eq1:=simplify(eq1,trig):
eq1:=convert(eq1,horner,[sin(teta),cos(teta)]):
```

This equation has 2 solutions 0, -0.8067834380 (or equivalently 5.47640187). Here we may use the general solving algorithm (section 2.3), the general solving algorithm with the gradient (section 2.4), the general solving algorithm with the gradient and Hessian (section 2.5) or the procedure devoted to trigonometric equation (section 2.13).

These algorithms are implemented in the test program `Test_Solve_General2`, `Test_Solve_Gradient_General2`, `Test_Solve_JH_General2`, while for the the specific procedure devoted to solving trigonometric equations we may use the program `Test_Solve_Trigo`. In the later case we may use or not the procedure which compute bounds on the roots of the equation.

We may then investigate the computation time when we have a pretty good idea of the solution. For example we assume that we have a root in the range [5.3, 5.7]. We may now use Newton method (see section 2.9) implemented in the test program `Test_Newton2`, Brent's method (see section 2.8 and Ridder's method (see section 2.7) implemented in the test program `Test_Ridder2`. An interesting point is that although Newton is faster than the others methods we may encounter convergence problem in this example. For example if the initial guess is 5.85 (which is closer to the solution 5.476 than the second solution 0) Newton will converge toward the 0 solution.

15.1.3 Example 4

This example is similar to example 1 except that we deal with the spatial case. The pose of the end-effector is defined by 6 parameters: the coordinates `x,y,z` of a specific point on the end-effector and three angles `p,t,h` which define its orientation. We have now 6 linear actuators connecting the end-effector and the ground. The length of the linear actuator are all equal to 60. When expressing the leg lengths as function of the parameters we may extract from the 6 equations a linear system in `x,y,z`. Solving this system and substituting in the remaining equations leads to a system of three equations in the unknowns `p,t,h`. The following Maple program enable to compute these equations.

```
with(linalg):
xa1:= -9:ya1:=9:xa2:=9:ya2:=9:
xa3:=12:ya3:=-3:xa4:=3:ya4:=-13:
xa5:=-3:ya5:=-13:xa6:=-12:ya6:=-3:
xb1:= -3:yb1:=7:xb2:=3:yb2:=7:
xb3:=7:yb3:=-1:xb4:=4:yb4:=-6:
xb5:=-4:yb5:=-6:xb6:=-7:yb6:=-1:

rot:=array([[cos(p)*cos(h)-sin(p)*cos(t)*sin(h)
,-cos(p)*sin(h)-sin(p)*cos(t)*cos(h),
sin(p)*sin(t)],[sin(p)*cos(h)+cos(p)*cos(t)*sin(h),
-sin(p)*sin(h)+cos(p)*cos(t)*cos(h),-cos(p)*sin(t)],
[sin(t)*sin(h),sin(t)*cos(h),cos(t)]]):

for i from 1 to 6 do
      OA.i:=array([xa.i,ya.i,0]):
      CBr.i:=array([xb.i,yb.i,0]):
      CB.i:=multiply(rot,CBr.i):
od:

cen:=array([x,y,z]):

for i from 1 to 6 do
      AB.i:=evalm(cen-OA.i+CB.i):
      ro.i:=dotprod(AB.i,AB.i,`orthogonal`):
      ro.i:=simplify(ro.i,trig):
od:
for i from 1 to 6 do eq.i:=ro.i-60^2: od:
er1:=eq2-eq1:er2:=eq3-eq1:er3:=eq4-eq1:er4:=eq5-eq1:er5:=eq6-eq1:
sw:=solve({er1,er2,er3},{x,y,z}):
assign(sw):

en1:=numer(eq1):en2:=numer(er4):en3:=numer(er5):

for i from 1 to 3 do
      en.i:=simplify(en.i,trig):
      en.i:=convert(en.i,horner,[sin(p),cos(p),sin(t),cos(t),sin(h),cos(h)]):
od:
```

We are looking for a solution in the domain:

$$[4.537856054, 4.886921908], [1.570796327, 1.745329252], [0.6981317008, 0.8726646262]$$

The system has a solution which is approximatively:

$$4.6616603883, 1.70089818026, 0.86938888189$$

Here we may use the general solving algorithm (section 2.3), the general solving algorithm with the gradient (section 2.4), the general solving algorithm with the gradient and Hessian (section 2.5).

These algorithms are implemented in the test program `Test_Solve_General`, `Test_Solve_Gradient_General`, `Test_Solve_JH_General` (in that later case Kantorovitch theorem is applied only at level 1).

15.2 Examples of applications of interval analysis

`ALIAS` offers tools that may be used to solve many problems and interval analysis has a broad range of application. In this section we will present examples of realistic problems that have been solved by interval analysis. Some of them have been solved using directly the solving algorithms of `ALIAS`, for others it has been necessary to write additional C++ code. Other examples may be found in the COPRIN web page.

15.2.1 A geometrical example

The following example originated from the INRIA Geometrica project. It exemplifies that focusing on the problem at hand while keeping in view that interval analysis will be used allows one to develop a very efficient algorithm that is at the same time faster and safer (in term of quality of the result) than conventional approaches.

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` to solve the system and uses the `Newton` procedure of ALIAS-Maple 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 arithmetics. 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 arithmetics.

15.2.2 A robot kinematics example

We present here a simplified version of the problem (see [10] for a more detailed version).

Consider a plane in the 3D space, called *the base*, on which lie 6 points $A_1, A_2, \ldots, A_6$ whose coordinates in an arbitrary reference frame.

Consider another plane, called the *platform*, on which lie similarly 6 points $B_1, B_2, \ldots, B_6$. To this plane we attach a frame, called the mobile frame. The coordinates of the 6 points B_i in the mobile frame are supposed to be known. These 6 points constitute a rigid body which has a position/orientation with respect to the reference frame (i.e the location of any point of the rigid body may be calculated as soon as we know the coordinates of a point of this body in the reference frame and the rotation matrix that relates the mobile frame to the reference frame).

Assume now that the distances between each pair A_i, B_i is fixed and is known. The problem is to determine what are the possible position/orientation of the rigid body such that these distance constraints are satisfied.

This problem is very important in practice. The platform is the "hand" of the robot and the distances between the A_i, B_i may be changed by using actuators. The only information we have in the robot is obtained through sensors that measure the distances between the A_i, B_i. Solving the above problem allows one to determine where is the hand of the robot by using the sensory information (but this problem has also other application such as robot calibration).

This problem is called the *direct kinematics* problem. It has a counterpart, the *inverse kinematics* problem: being given the position/orientation of the rigid body computes the distances between the pairs A_i, B_i. This inverse kinematics problem has a straightforward solution: each squared distance can be calculated as a function of the parameters that describe the position/orientation of the rigid body. Hence solving the direct kinematic problem is equivalent to solve the system of equations of the inverse kinematics in the position/orientation parameters.

This problem is considered as one of the most difficult robot kinematics problem. Before 1992 the only known method to solve this problem was to use the Newton scheme and there was no theoretical result on this problem. Afterward with the use of sophisticated mathematics it has been possible to show theoretical results:

- there will be at most 40 solutions, complex and real
- examples with 40 real solutions have been exhibited
- it is possible to reduce this problem to the the solving of a 40th order univariate polynomial

But still was lacking a practical algorithm to compute the solutions: indeed the manipulation needed to get the 40th order polynomial are difficult to automatize.

The fastest algorithm to solve this problem is the use of the Gröebner basis package of Faugère, Rouillier that allow to compute all the solutions in a computation time that ranges between a few seconds to less than 30 seconds.

Interval analysis allows to solve this problem. Basically we have noticed that we may choose as unknowns the coordinates of 3 points on the platform (called the *reference points*). Indeed the coordinates of the 3 other points may be obtained as linear combination of the 3 chosen points (i.e. they are virtual points). Furthermore the only equations we have are only distance equations: distances between the pairs A_i, B_i and distances between the reference points and the virtual points. Hence the distance solving algorithm described in section 2.16 is convenient for this problem.

Trials have shown that the computation time ranges between 10 to 40 seconds, which is competitive with the algorithm of Faugère, Rouillier. Furthermore in most practical cases we are not interested in computing *all* the solutions but only the one that corresponds to the actual position/orientation of the platform and as the direct kinematics problem has to be solved almost continuously and the robot speed is limited we may determine a small search domain in which the current solution should be located. In that case the interval analysis based algorithm is very fast as it is sensitive to the search space (at the opposite of the Gröebner approach that has to compute all the solutions and then sort which one is the current solution and hence is working in a constant time). The algorithm is also safe compared to the Newton scheme that may converge toward a solution that is not the current one. Indeed we may guarantee that the current solution is included in the algorithm result. Furthermore if more than one solution is found then it is better to stop immediately the robot as the result of the calculation is used to control it.

15.2.3 A robot control problem

The problem presented here is for a robot described in the previous section. We have seen that the hand of the robot was moving with respect to its base by using linear actuators that allow to change the distance between the pairs of points A_i, B_i. But in practice these actuators have a limited stroke and consequently the distance ρ between the points has to lie within a given range:

$$\rho_{min} \leq \rho \leq \rho_{max}$$

Furthermore the actuator is attached at point A_i, B_i by joints (typically ball-and-socket or universal joints) that have limited motion. Hence the platform may reach only a limited set of position/orientation that is called its *workspace*.

Assume now that the hand has to follow a time-dependent trajectory: it is clearly important to verify that all the points of this trajectory lie within the workspace of the robot. The trajectory is defined in the following manner: the position/orientation parameter are written as analytic function of the time T (which is assumed to lie in the range [0,1] without lack of generality). Using the solution of the inverse kinematics (see the previous section) it is then possible to express the distance ρ as functions of the time. For the trajectory to lie in the workspace we have to verify the 12 inequalities:

$$\rho_{min} \leq \rho(T) \leq \rho_{max}$$

when T lie in the range [0,1]. As the analytical form of the position/orientation parameters may be arbitrary we are looking for a generic algorithm that may deal with such arbitrary trajectory.

This can easily be done with interval analysis (see [9] for a detailed version). First we define the trajectory in Maple and compute the analytical form of $\rho(T)$ (and of any other constraint that may limit the workspace of the robot). We get a set of inequalities that has to be satisfied for any T in [0,1] if the trajectory lie within the workspace. The analytical form of these inequalities are written in a file: this allow their interval evaluation for any T range by using the `ALIAS` parser. Then the general solving procedure of `ALIAS` may be used to determine if there is a T such that at least one constraint is violated.

An important point is that the algorithm allow to deal with the uncertainties in the problem. A first uncertainty occurs when controlling the robot along its nominal trajectory. Indeed the robot controller is not perfect and there will be a positioning error: for a nominal value of the position/orientation parameters $\mathbf{X}$ the reached pose will be $\mathbf{X} + \Delta\mathbf{X}$ where $\Delta\mathbf{X}$ can be bounded. A second uncertainty source is due to the differences between the theoretical geometrical model of the robot and its real geometry. Indeed to solve the inverse kinematics we use the coordinates of the A_i in the reference frame and of the B_i in the model frame. In practice however these coordinates are known only up to a given accuracy: hence these coordinates for the real robot may have any value within given ranges. Hence the inequalities of the problem have not fixed value coefficients but interval coefficients. But this no problem for interval analysis as the general solving procedures may deal with such inequalities. Hence if the algorithm find out that all inequalities are verified for any T in [0,1], then this means that whatever is the real robot and the positioning error of the robot controller the trajectory followed by the robot will fully lie within the robot workspace.

15.2.4 Robot singularity analysis

In the robot presented in the previous sections we have seen that the inverse kinematic relates the the distances ρ between the A_i, B_i to the position/orientation parameters of the platform $\mathbf{X}$ by $\rho = F(\mathbf{X})$. An important matrix is the jacobian matrix of this relation $J = ((\partial F/\partial X_i))$. Indeed the forces τ in the legs A_iB_i are related to the forces/torques $\mathcal{F}$ applied on the platform by

$$\mathcal{F} = AJ\tau$$

where A is a matrix that is never singular. Hence for given forces/torques applied on the platform the force in each leg may be computed as a ratio whose denominator if $|AJ|$. A configuration where $|J| = 0$ is called a *singularity* of the robot. In this configuration the forces in the leg of the robot may go to infinity, causing a breakdown of the robot. Hence a very important practical problem is to be able to determine if there is a singularity within a given workspace $\mathcal{W}$ of the robot.

Usually this workspace is defined in the following manner: a geometrical object that will include all possible location of a specific point of the platform and ranges for the parameters that describe the orientation of the platform.

The determinant of J can usually be expressed as a function of the position/orientation parameters. Interval analysis is appropriate to design an algorithm that check if there is a singularity within a given workspace. For that purpose we will take a point inside the workspace and calculate the value of $|J|$ at this point. Let us assume that this value is positive. A singularity will occur within the workspace if we are able to find another point in the workspace where $|J|$ will be negative.

We may assume that we are able to calculate a bounding box of the workspace and that we are able to design a test that allow to determine if a box for the position/orientation parameters is fully included in the workspace, is fully outside or part of it is inside and part of it is outside.

With that it is easy to design an algorithm that determine if the workspace includes a singularity (see [12] for more details).

First we use interval arithmetics to determine bounds for the value of $|J|$. If the lower bound of the interval is positive is positive, then there is no singularity within the workspace. Otherwise we bisect the box. For the new boxes we first test if the box is inside the workspace: if one box is fully outside the workspace we discard it. If a box is not fully inside we bisect the box. For a box that is fully inside we compute the interval evaluation of $|J|$. If the lower bound is positive we discard the box. If the upper bound is negative, then for any position/orientation defined by parameters within the box ranges the determinant will be negative: hence the workspace includes a singularity and we may stop the algorithm. If all the boxes in the list have been processed, then there is no singularity within the workspace.

Note that it is easy to generalize the algorithm so that uncertainties in the robot geometry are taken into account. A general version of this algorithm has been implemented using `ALIAS` and Maple and is able to deal with different robot architectures. Maple is used to compute the analytical form of $|J|$. The `ALIAS` parser is used to perform the interval evaluation of the determinant.

15.2.5 Robot workspace analysis

For the robot presented in the previous examples we have a set of parameters $\mathbf{X}$ that allow to represent the position/orientation of the platform. At the same time we have various geometrical parameters that describe the possible geometry of the robot. Examples of the these parameters are the distance ρ between the A_i, B_i, the angle θ_i between the vector $\mathbf{A_iB_i}$ and a fixed direction in space (this angle represent the amplitude of the motion of the joint we have at A_i).

In practice all the geometrical parameters are submitted to constraints. For example the distances ρ must satisfy $\rho_{min} \leq \rho \leq \rho_{max}$, the angle θ_i must be such that $|\cos\theta_i| = |\mathbf{A_iB_i}.\mathbf{n_i}/\rho_i| \geq \cos\theta_d$ where $\mathbf{n_i}$ is a unit vector that describes the main orientation of the ball-and-socket joint at A_i and θ_d is the maximal possible rotation of this joint. All these constraints imply that the value of the elements of $\mathbf{X}$ are restricted to lie in a specific region embedded in the 6-dimensional space, this region being called the *reachable workspace* of the robot as the points of this region define all the position/orientation that can be reached by the platform.

In some cases it is possible to find an analytical description of the workspace (for example if the orientation is fixed and only the distance constraints are taken into account) but as soon as all the constraints are taken into account such analytical form is difficult to obtain. Interval analysis allows to obtain an approximation of the workspace as a set of 6-dimensional boxes. It is only an approximation as the algorithm, based on a classical bisection method, discard boxes whose width is lower than a given threshold. At the same time it is possible to obtain a bound on the error between the volume of the approximation and the volume of the workspace. For more details see [11]. A variant of the algorithm allows to solve efficiently a very practical problem: is a given region included in the reachable workspace of the robot ?

15.2.6 Robot synthesis example

The robot presented in the previous sections has numerous advantages but its performances are very sensitive to its geometry (i.e. to the location of the points A_i, B_i). The previous sections have presented *analysis* problems: analyze for a given robot what are its performance. But we may also consider an even more complex problem which is the *synthesis* problem i.e. find what should be the geometry of the robot such that it satisfies some performance criteria. This is very complex issue but one which has clearly a large impact in practice (for example when designing a robot whose load will be over 2 tons and whose accuracy should be better than a micrometer as considered by the European Synchrotron Radiation Facility in Grenoble).

Consider for example the following problem: what should be the geometry of the robot (i.e. the location of the A_i, B_i and the limits ρ_{min}, ρ_{max} on the ρ) such that a given set $\mathcal{W}$ of position/orientation $\{X_1, \ldots, X_n\}$ can be reached by the platform ?

The inverse kinematic analysis allows to obtain the constraints $\rho_{min} \leq \rho \leq \rho_{max}$ as a set of inequalities $F(\mathcal{P}, X_i) \leq 0$ between the design parameters $\mathcal{P}$ and the elements of the set $\mathcal{W}$. A classical approach to solve this problem will be to determine the set $\mathcal{P}$ that minimize the cost function $\sum_i F(\mathcal{P}, X_i)$. But this approach has many drawbacks:

- we cannot guarantee to find even one solution
- it will give at most a few solutions to the synthesis problem while in general there will be an infinite number of solutions
- for a given solution the geometry of the real robot will be different from the theoretical solution due to the manufacturing tolerances. Hence we cannot guarantee that the real robot will really be able to reach the set $\mathcal{W}$

Interval analysis is a more appropriate approach. First of all it is reasonable to assume that the geometrical parameters are bounded as they define the overall size of the robot. We may thus define a box as a set of ranges, one for each design parameters. Using interval arithmetics we are then able to compute the lower and upper bounds of each F for each element of $\mathcal{W}$. The different cases that may occur for a box are:

- the lower bound of the interval for one F is positive: we discard the box
- the upper bound of the intervals for all F is negative: the box is stored as a solution
- the lower bound of the intervals for all F is negative while their upper bound is positive: we bisect the box at the variable having the largest range except if all ranges have a width that is lower or equal to twice the manufacturing errors, in which case we discard the box

Using this approach we get not only one solution but ranges for the solution: indeed each point of the solution boxes satisfy all inequalities F which means that the theoretical robot will be able to reach all the elements of $\mathcal{W}$. But we may take also the manufacturing errors ϵ into account. Indeed for a range A=$[\underline{x}, \overline{x}]$ in a solution box we may consider the range AP=$[\underline{x} + \epsilon, \overline{x} - \epsilon]$ that is included in the previous range and exists as the ranges of a solution box have a width at least 2ϵ. If we choose as design parameter a point in AP, then for the real robot the design parameter value will be included in A. As this is true for all the design parameters, then we may guarantee that the real robot will be able to reach all elements of $\mathcal{W}$.

Using this approach we are able to find all the geometries such that the real robot satisfy the workspace constraints. Now we may consider another design criteria and use the same approach, that will lead to another set of design parameters. Taking the intersection of this set with the set obtained for the workspace we will be able to compute all robot geometries that satisfy the new design criteria **and** the workspace constraint.

Here again interval analysis is an elegant approach that allows to solve a very practical problem. This approach has been used by the COPRIN project in various industrial contracts.

15.2.7 A quantum mechanics example

The Kochen-Specker theorem is one of the more important in quantum mechanics. Its a application involves the solving of the following problem (that has been simplified for the sake of clarity).

Consider n unit vectors named `1,2,..9,A,B,..Z` in a four dimensional space. A *group* of vectors is a set of 4 vectors that have to be orthogonal to each other and not identical or opposite. A `m-sequence` is a set of `m` groups such that:

- each vector in the sequence has the color black or white
- in each group of the sequence there must be exactly 3 black vectors and one white vector

The smallest known sequence is a 9-sequence named the Cabello's system:

`1234,4567,789A,ABCD,DEFG,GHI1,35CE,29BI,68FH`

which involves 18 vectors.

Our objective is to to make an exhaustive search of m-sequences. For that purpose we use a software designed by B.D. McKay, called the *generator*, that provides for any set of `a` vectors and any sequence with `b` groups all the `b`-sequences that satisfy the color constraints and are not isomorphic (for example changing the name of the vectors will lead to the same sequence). Then we consider each potential `b`-sequence and we must check if there are at least one set of coordinates for the vectors such that the orthogonality constraints for each group are satisfied together with the non-collinearity between the vectors.

Such verification is not so easy. Indeed the orthogonality constraints for a group involves 6 equations: consequently for a `b`-sequence with `a` vectors we have 6`b` orthogonality equations, `a` equations that indicate that the vectors are unitary. If we consider for example the Cabello's system the number of equations is therefore already 74 for 72 unknowns. Although the problem is algebraic the solving method in algebraic geometry have difficulties to handle such number of equations (for a sequence that is one of the smallest).

A second problem is that an exhaustive enumeration of all potential `b`-sequences leads to a very large number of sequences. For example for `a`=10 and `b`=12 the number of sequences is 197 885 058 ...

Nevertheless the problem was convenient for interval analysis. Indeed the unknown coordinates are component of unitary vectors and hence will lie in the range [-1,1]. Furthermore if a vector is solution then its opposite is also solution: hence we may restrict one of the coordinates of the vector to lie in the range [0,1]. In the same manner any rotation applied simultaneously to all solution vectors will lead to another set of solution vectors. Hence we may choose arbitrary any group so that its vectors constitutes an orthogonal basis of the four dimensional space. The coordinates of the vectors in this group will then be (1,0,0,0), (0,1,0,0), (0,0,1,0) and (0,0,0,1). Furthermore if one of these vectors appears in another group, then the orthogonality constraint will imply that one coordinate of all vectors of this group (except the basis vector) will be 0.

To solve this problem we have then developed a first very fast interval analysis algorithm that has been incorporated in the generator and eliminates sequence that trivially will have no solution. For example a sequence starting with `1234,1235` will be eliminated as `5` should be `4` or `-4`. This algorithm drastically reduces the number of generated sequences. Then for the remaining sequences we have used a general solving procedure of `ALIAS`. Even with the above simplification the size of the systems are large: for Cabello's system we have reduced the problem to 32 unknowns and 82 equations that can be solve in about 60 seconds on a laptop. But eliminating sequence without solution is usually faster. For example the system

`1234,1256,1378,129A,24BC,34DE,5DFG,7HIJ,8BFK,9ADE,CGHI,6EJK`

with 59 equations and 26 unknowns is eliminated in 10 seconds.

Although this work has not been completed yet we have already shown that Cabello's system is the smallest sequence (all sequences with a smaller number of groups have no solution).

Chapter 16

Troubleshooting: ALIAS does not work!

16.1 Compilation problems

Compilation problems may occur:

1. if you are using the Gnu C++ compiler with a version higher than 2.95 (the code of `BIAS/Profil` has not yet been modified to accommodate the new standard)

2. if you try to compile the interval evaluation of very complex expressions. Typically you may get a message indicating that the compiler has exhausted its virtual memory. C++ seems to be indeed quite sensitive to the size of the expressions.

To solve problem 2 a first possibility is to turn off the optimization flag that is used for the compilation and instead use `-g` as the compilation flag.

If the compilation cannot still be completed you are confronted to a difficult problem. Most probably the compilation problems are due to the size of the expressions that you are manipulating for your problem. A first solution is to try to simplify the generated expression using auxiliary variables for terms that appear at least twice in the code. ALIAS-Maple proposes a mechanism that may allow to perform part of this task automatically. But many time it will be still necessary to edit by hand the C++ program. If you are using ALIAS-Maple the procedure involving the expressions, their jacobian and Hessian are named 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 the files that involves the interval evaluation of your expressions that you may have included directly in your main program.

This is the case for the C++ code produced by ALIAS-Maple which include the expressions files in the main program. Let us consider the example of the general solving procedure. The main program is named `_GS_.C` and is compiled with the makefile `_makefile`. This main program includes the expressions evaluation file `_F_.c` and eventually the jacobian and hessian evaluation files (`_J_.c , _H_.c`). To compile separately the expressions evaluation file first rename this program `_F_.C` and 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"
```

and remove the `#include "_F_.c"` in the main program `_GS_.C`.

Then the makefile should 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
```

If the previous method does not work, then there is an alternative that will always work but at the expense of an increased computation time: instead of compiling your expression you may rather use the ALIAS-C++ parser to interpret all the expressions (see chapter 12). Basically you will have to write the analytical form of each expression you will have to evaluate in a text file and execute specific procedures of the parser library that will perform the interval evaluation of these expressions by reading this text file. Note that ALIAS-Maple is able to produce code that performs this operation (see the `MakeF, MakeJ, MakeH` procedures.

16.2 Execution problems

16.2.1 Interval valuation problems

You have now successfully completed the compilation of your program and you run it. But it just crashes with an error message such as:

```
BIAS Error: ArcSin argument out of range
BIAS Error: Power: Negative or zero exponent with zero base
BIAS Error: Division by Zero
```

You have to remember that not all expression may have an interval evaluation according to the ranges of the unknowns (see section 2.1.1.3 for a complete list of the expression that may lead to an evaluation problem.

To solve this problem first consider the expressions you have, determine which of them may lead to an evaluation problem and examine if you may modify the expression to avoid the interval valuation problem (for example for an equation whose terms have a common denominator just consider the numerator of the expression as the interval evaluation of the denominator may include 0, a typical case for which an interval valuation cannot be calculated).

If the problem persist you will have to identify which terms may lead to an interval valuation problem for each expression (we will call them the *faulty terms*). Then before performing the interval evaluation of the expression you will have to compute the interval evaluation of the faulty terms. If this evaluation leads to interval that will cause an evaluation problem, then instead of performing the interval evaluation of the expression you will affect an interval to the expression that depends on the context.

For example assume that you have to solve the equations system

```
y/(x-1)+cos(y)=0
arcsin(y)+x*y-1=0
```

with `x, y` in the range [-10,10]. The faulty term for the first equation is `x-1` whose interval evaluation cannot include 0 (as we divide by `x-1`). The faulty term for the second equation is `y` which must lie in the range [-1,1] (as we have `arcsin(y)`). Now you want to write the procedure that will be used to compute the interval evaluation of these equations that is to be used by the general solving procedure. Initially you will have written this procedure as:

```
INTERVAL_VECTOR F(int l1,int l2, INTERVAL_VECTOR &X)
{
INTERVAL_VECTOR V(2);
```

```
if(l1==1)
    V(1)=X(2)/(X(1)-1)+Cos(X(2));
if(l2==2)
    V(2)=ArcSin(X(2))+X(1)*X(2);
return V;
}
```

In that case the procedure will crash immediately as for the initial search domain the first equation cannot be evaluated. Now if the interval evaluation of the faulty term of the first equation includes 0 we must not evaluate the equation. But still we wish to eliminate a box for which the interval evaluation of `x-1` includes 0 only if the maximal absolute value of the interval evaluation is very close to 0. We may deal with this problem with the following code:

```
if(l1==1)
    {
    U=X(1)-1;
    if(Sup(IAbs(U))<1.e-4)V(1)=1;
    else
        {
        if(Inf(U)<=0 && Sup(U)>=0)V(1)=INTERVAL(-1.e8,1.e8);
        else V(1)=X(2)/U+Cos(X(2));
        }
    }
```

If the interval evaluation of `x-1` has a maximal absolute value lower than 1e-4 the interval evaluation of the first equation is 1 and the solving algorithm will discard the corresponding box. On the other hand we set the interval evaluation of the equation to an arbitrary interval that includes 0 so that the solving algorithm will not discard the box.

For the second equation the management of the evaluation problem is somewhat different. If the interval of `y` has no intersection with the interval [-1,1] then we may discard the box. Otherwise we may change the the interval for `y` to this intersection and proceed with the evaluation of the equation. To compute this intersection we use the `Intersection` procedure of `BIAS/Profil` that returns 0 if two intervals have an empty intersection. The code is

```
if(l2==1)
    {
    if(Intersection(U,X(2),INTERVAL(-1,1))==0)V(2)=1;
    else
        {
        X(2)=U;
        V(2)=ArcSin(X(2))+X(1)*X(2);
        }
    }
```

Clearly managing this process by hand is tedious and prone to errors. ALIAS-Maple offers a procedure that determine all the constraints that must be satisfied by the unknowns for the expressions to be interval-valuable and another procedure that manages automatically the process we have performed in the example.

16.2.2 Wrong results

If ALIAS-C++ returns a wrong result a first thing to do is to check he routines that are specific to your problem. A typical mistake is to write expression such as `2/3*x+1/3*y..` that are evaluated to 0 whatever is `x, y` (such expression should be written as `2./3.*x+1./3.*y`). Otherwise see the bug section.

16.2.3 Running out of memory

Even for very large problem any program generated with `ALIAS`-C++ should not run out of memory as soon as:

- the single bisection mode is chosen by setting `Single_Bisection` to a value larger than 0
- the reverse storage mode is chosen by assigning `Reverse_Storage` to a value larger than the number of variables.

However in some cases some single bisection mode may lead to run out of memory: in that case set the bisection mode to 1.

For very large problems you may run into memory problems if you allocate a large amount of memory to the ALIAS storage. Try to decrease the storage size to just allocate the amount of memory necessary for the algorithm. If you are using the solving procedures with jacobian you may avoid storing the jacobian of all boxes by setting the flag `ALIAS_Store_Gradient` to 0.

16.2.4 Large computation time

16.2.4.1 Changing the formulation of the problem

Interval analysis is a very specific domain: 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 depending 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.

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

The rule for the right formulation is to find the right compromise between the complexity of the expressions (very complex expressions will usually lead to a large overestimation of the expressions) and the number of variables.

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 7.4).

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 (see for example the evaluation of the Wilkinson polynomial in section 5.9.3.2). 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 a 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 bench). 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 will be 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.

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

16.2.4.2 Choosing the right heuristics

In interval analysis we use a lot of heuristics (such as the use or not of the derivatives, the use of the 2B, 3B filtering methods, ...) and finding the combination that will be the most efficient to solve a problem is a complex issue. Furthermore the choice of parameters of these heuristics (for example the step size in the 3B method) may have a very large influence on the computation time (and we mean a *really* large influence with a decrease factor of the computation time that may be 10^4).

In our experience for solving equation systems you have better always use the 2B and 3B methods (see sections 2.3.2 and 2.17). But this manual provides also numerous filters that may be more specific but also very efficient.

16.2.5 Crash and bugs

You may have found an `ALIAS` bug either because the program crashes or because you get a wrong result (or no result at all while there is clearly at least one solution) or because the computation time is very large.

In the first case first check the error message you get (if any). If you get a message that originates from `Bias/Profil` such as `Abort: Division by Zero` this means that you have an expression whose interval evaluation is not allowed in some cases: see section 16.2.1 to determine how to avoid this problem. In the second case check with section 16.2.2 that you have correctly written your expressions. In the third case see if some method proposed in section 16.2.4 may help.

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

The bug/problem 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.

16.3 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:

```
Debug_Level_Solve_General_Interval=0;
Debug_Level_Solve_General_Interval=2;
```

or, if you are running ALIAS from MAPLE, add to your MAPLE program one of the following line:

```
`ALIAS/debug`:=2:
`ALIAS/debug`:=0:
```

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 indicate that the algorithm is dealing with the 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 informations you may figure out what is going on in the algorithm.

Contents

Bibliography

[1] Coleman R. La méthode de weyl pour le calcul simulatané des racines d'un polynome. Research Report 881-M, IMAG, April 1992.

[2] Collavizza M., F. Delobie, and Rueher M. Comparing partial consistencies. *Reliable Computing*, 5:1–16, 1999.

[3] Démidovitch B. and Maron I. *Eléments de calcul numérique*. Mir, 1979.

[4] Durand E. *Solution numérique des équations algébriques*. Masson, 1960.

[5] Hansen E. *Global optimization using interval analysis*. Marcel Dekker, 1992.

[6] Jaulin L., Kieffer M., Didrit O., and Walter E. *Applied Interval Analysis*. Springer-Verlag, 2001.

[7] Kearfott R.B. and Manuel N. III. INTBIS, a portable interval Newton/Bisection package. *ACM Trans. on Mathematical Software*, 16(2):152–157, June 1990.

[8] Krawczyk R. Newton-algorithmen zur bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[9] Merlet J-P. A parser for the interval evaluation of analytical functions and its applications to engineering problems. *J. Symbolic Computation*, 31(4):475–486, 2001.

[10] Merlet J-P. Solving the forward kinematics of a Gough-type parallel manipulator with interval analysis. *Int. J. of Robotics Research*, 23(3):221–236, 2004.

[11] Merlet J-P. Determination of 6D workspaces of Gough-type parallel manipulator and comparison between different geometries. *Int. J. of Robotics Research*, 18(9):902–916, October 1999.

[12] Merlet J-P. and Daney D. A formal-numerical approach to determine the presence of singularity within the workspace of a parallel robot. In F.C. Park C.C. Iurascu, editor, *Computational Kinematics*, pages 167–176. EJCK, Seoul, May, 20-22, 2001.

[13] Mignotte M. *Mathématiques pour le calcul formel*. PUF, 1989.

[14] Mineur H. *Technique de calcul numérique*. Dunod, 1966.

[15] Miranda C. Un' osservatione su un theorema di Brouwer. *Bulletino Unione Mathematica Italiana*, pages 5–7, 1940.

[16] Moore R.E. A test for the existence of solution to nonlinear systems. *SIAM J. of Numerical Analysis*, 14:611–615, 1977.

[17] Moore R.E. *Methods and Applications of Interval Analysis*. SIAM Studies in Applied Mathematics, 1979.

[18] Neumaier A. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[19] Neumaier A. *Introduction to Numerical Analysis*. Cambridge Univ. Press, 2001.

[20] Neumaier A. and Merlet J-P. Solving real-life robotics problems with interval techniques. In *SIAM Workshop on Validated Computing*, page 148, Toronto, May, 23-25, 2002.

[21] Ratscheck H. and Rokne J. Interval methods. In Horst R. and Pardalos P.M., editors, *Handbook of global optimization*, pages 751–819. Kluwer, 1995.

[22] Rex G. and Rohn J. Sufficient conditions for regularity and singularity of interval matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(2):437–445, 1998.

[23] Stiefel E.L. *An introduction to numerical mathematics*. Academic Press, 1963.

[24] Van Hentenryck P., Michel L., and Deville Y. *Numerica: A Modeling Language for Global Optimization*. The MIT Press, 1997.

[25] Yamamura K., Kawata H., and Tokue A. Interval solution of nonlinear equations using linear programming. *BIT*, 38(1):186–199, 1998.

[26] Zippel R. *Effective polynomial computation*. Kluwer, 1993.

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‘`.

Index

B