W. Krämer, U. Kulisch, R. Lohner

Numerical Toolbox for Verified Computing II

Advanced Numerical Problems

This is a draft version which still contains errors.

Preface

To L. Kronecker we owe the remark, "God created the natural numbers, all else is man made." This difference is reflected in computers, where arithmetic on the natural numbers may be performed exactly, while for all else, only approximately.

Traditionally in Scientific Computing, the computer is only capable of computing approximate solutions to a problem. Even though approximations are often satisfactory, they may nevertheless be inaccurate or even completely wrong. With traditional floating-point-arithmetic, it is usually very difficult to tell when something has gone wrong simply by looking at the numerical results.

In contrast, this treatise aims to do rigorous mathematics with the computer using floating-point numbers. For that purpose we enhance the mathematical power of the digital computer by defining and using the ideal computer. Instead of reducing all calculations to the four basic arithmetic operations for floating-point numbers we require twelve fundamental data types or mathematical spaces with operations of highest accuracy in a computing environment. We also require a number of elementary functions of highest accuracy for four basic data types. In case the available real computer differs from the required ideal computer the missing operations can be simulated be use of the available integer arithmetic. Programming environments which provide these operations (the XSC-languages) are available.

The twelve fundamental data types or mathematical spaces of the ideal computer consist of the four basic data types real, complex, interval and complex interval and the vectors and matrices over these types. All computer operations for these types are directly defined by the concept of "semimorphism":

If M is any one of these twelve spaces and N its computer-representable subset, and if $\circ$ is any arithmetic operation in M, then the corresponding computer operation $\boxdot$ in N is defined by

$$(RG) \quad a \boxdot b := \square(a \circ b) \quad \text{for all } a,b \in N$$

where $\square : M \to N$ is a mapping from M onto N, called a rounding, with the following properties:

$$\begin{array}{llll} (R1) & \square(a) = a & \text{for all } a \in N & \text{(projection)} \\ (R2) & a \leq b \Rightarrow \square(a) \leq \square(b) & \text{for } a,b \in M & \text{(monotonicity)} \\ (R3) & \square(-a) = -\square(a) & \text{for all } a \in M & \text{(antisymmetry)} \end{array}$$

This means that every computer operation is performed in such a way that it produces the same result as if the mathematically correct operation were first performed and the exact result then rounded into the computer-representable subset. This rule is optimal in the sense that there is no better computer-representable approximation to the true result with respect to the prescribed rounding. $(R3)$ imposes the property of symmetry on N, i. e. if $a \,\epsilon N$, then $-a \,\epsilon N$.

For the interval spaces among our twelve fundamental data types the order relation in $(R2)$ is the subset relation. A rounding from any interval set M onto its computer-representable subset N is defined by properties $(R1), (R2)$ (with $\leq$ replaced by $\subseteq$), plus the additional property

$(R4) \quad a \subseteq \Box(a) \quad$ for all $a \,\epsilon M \quad$ (inclusion)

These interval roundings are also antisymmetric, that is, they satisfy $(R3)[K], [KM]$.

This definition of Computer Arithmetic by the concept of semimorphism should be used as an axiomatic definition of computer arithmetic in the context of programming languages. This is the case in the XSC languages.

A careful analysis of the requirements of semimorphisms is given in $[K], [KM]$. Many properties of both the order structure and the algebraic structure are invariant under a semimorphism. For more details, refer to $[K], [KM]$. All semimorphic computer operations are of 1 ulp accuracy. 1/2 ulp is achieved in the case of rounding to nearest. It is ultimately shown that semimorphic operations for the twelve fundamental data types of the ideal computer can be provided if, on a low level, preferably in hardware, 15 fundamental operations are available: the five operations $+, -, *, /, \bullet$, each one with the three roundings $\Box, \bigtriangledown, \bigtriangleup$. Here $\bullet$ means the dot procuct of two vectors, $\Box$ is a monotone, antisymmetric rounding, e. g. rounding to nearest, and $\bigtriangledown$ and $\bigtriangleup$ are the monotone downwardly and upwardly directed roundings from the reals into the floating-point numbers. All 15 operations $\circ \epsilon \{+, -, *, /, \bullet\}$ are defined by (RG).

These 15 fundamental operations are found in a variety of programming systems [PASCAL-XSC, ACRITH-XSC, C-XSC, FORTRAN-XSC]. Twelve of the 15 operations are provided by the IEEE standards $[,], \circ \epsilon \{+, -, *, /\}$. These standards also prescribe specific data formats. By adding just three more operations, all arithmetic operations in the basic product spaces of mathematics can be performed with 1 or 1/2 ulp accuracy in each component. The concept of semimorphism is not bound to specific data formats.

In Numerical Analysis, the dot product is ubiquitous. It appears in complex arithmetic, vector or matrix arithmetic, multiple precision arithmetic, and iterative refinement techniques, among many other places.

The arithmetic of the ideal computer has been implemented in hardware, in firmware, and in software on many different platforms and has been available since the early 1980's. The accumulation of products in a long fixed-point register allows an easy realization of the optimal dot product as defined by (RG). Besides being always accurate, fixed-point accumulation in hardware turns out to be at least as fast as a customary implementation using floating-point operations. Several intermediate steps of the latter as composition, decomposition, normalization, and rounding of floating-point values as well as unnecessary load and store operations become gratuitous, if accumulation is performed into a fixed-point register.

Intervals bring the continuum into the computer and open a new dimension in Numerical Analysis. An interval between two floating-point numbers describes the continuum of real numbers between these floating-point bounds. Roughly speaking, interval arithmetic brings guarantees into computation while the scalar or dot products deliver high accuracy. The mathematical insight into the fundamental principles of computer arithmetic of the ideal computer and the technology to realize them in hardware have been available for many years and should be common knowledge by now. The arithmetic of the ideal computer makes numerical computations more reliable and allows computers to do real mathematics.

In this volume a number of standard problems of Numerical Analysis are treated. All methods, algorithms and routines provide highly accurate verified enclosures of the solution or terminate with the answer that no correct solution could be found. In case of success in general even existence and uniqueness of the solution is established by the algorithm.

Among the problems that are dealt with are linear systems of equations with real, complex, interval and complex interval coefficients, sparse linear systems, eigenvalue problems, computation of the matrix exponential, numerical quadrature, staggered arithmetic, automatic differentiation and Taylor arithmetic, polynomials of several variables, and arithmetic for long reals and long real intervals. For all these algorithms the arithmetic of the ideal computer, as discussed above is essential. Although all algorithms are well designed and based on long experience the main purpose of this book is to provide sample solutions and to stimulate further research.

By way of example we discuss two of the methods that are described in the following chapters a little more closely. This demonstrates that the use of the ideal computer and appropriate programming environments are essential.

Staggered arithmetic is a very elegant way providing a long real or long real interval arithmetic. A variable of type staggered is just an array of reals, the sum of which does not fit into a single floating-point number. In the ideal case the components of the array do not overlap digitwise. Addition, subtraction and accumulations of staggered variables are performed by adding them into the long accumulator (which is a fixed-point register). Multiplication is just a dot product. Division is executed through a simple iterative scheme. A local variable rules the precision (or number of components) of variables of type staggered. This local variable can be enlarged or decreased at any time during a calculation. Thus the working precision of a program part can easily be adapted to obtain a desired accuracy of the result. The computation of the matrix exponential in chapter 6 is based on staggered arithmetic.

Staggered arithmetic is a very fast way to perform a long real arithmetic if the computation of dot products is supported by hardware. It can be used to speed up many programs of computer algebra.

As a second example we mention numerical quadrature. Most sophisticated methods have been developed in Numerical Analysis to control the step size by so called adaptive strategies. Traditionally all these methods are based on error estimates which are obtained by comparing computed approximations of different quadrature formulas. The 15th rule in case of Simpson's method is the simplest such test. All these methods may easily fail. They depend essentially on the reliability of the computed approximations.

Validated numerical quadrature begins with an evaluation of the remainder term of the quadrature formula. Depending on its size, the order of the method, the step size and/or the working precision can now be adapted. Evaluation of the remainder term needs the arithmetic of the ideal computer and appropriate programming support. All this is available for instance under PASCAL-XSC. Enclosures of the remainder term are obtained by automatic differentiation or Taylor-arithmetic. These techniques are discussed in chapters 4 and 7. The corresponding operators are somehow part of the runtime system of PASCAL-XSC. The user just has to specify his variables as being, for instance, of type interval taylor. Then the compiler computes enclosures of the taylor coefficients out of the expresssion for the function. As soon as the remainder term is under a given accuracy threshold the quadrature formula is evaluated in interval arithmetic. This is just an interval dot product. The sum of both gives a validated enclosure of the integral.

Karlsruhe, June 1994 The Authors

Table of Contents

Chapter 1

Introduction

1.1 On the Purpose of Result Verification

Classical numerical methods programmed on a computer in floating-point arithmetic deliver approximations to solution(s) of the problems which they are supposed to 'solve'. It lies in their approximative nature that they cannot even address the question whether the problems they treat have a solution at all – or none or even several. In this sense these methods, or at least the corresponding computer programs, are strictly speaking only heuristic and their results do not carry any mathematical information – only hints at best.

The reasons for this situation are obviously an imprecise arithmetic on the computer – i.e. rounded floating-point arithmetic with a *finite* set of numbers only – and the additional errors which are introduced by many numerical methods like truncation errors due to stopping an iteration, discretization errors in the treatment of differential or integral equations and the like. The combination of all these errors seems to make it impossible to obtain rigorous mathematical results from a computer program where all or part of these errors come into play.

Only computer programs which use some sort of exact arithmetic like integer or rational arithmetic and which do not commit any truncation or discretization error, e.g. by carrying out symbolic computations like in computer algebra systems, are commonly believed to produce mathematically rigorous results.

Among numerical analysts and users of numerical software it is unfortunately not common knowledge that methods and computer programs which contain these sources of errors may nevertheless produce mathematically *rigorous* results. The reason for this is that floating-point arithmetic can be defined in a mathematically clean way which includes precisely defined roundings and thus the result of floating-point computations is just as reliable as that of integer or rational computations. However, in contrast to the latter, the 'exactness' of such a precisely defined floating-point arithmetic is not that it computes the exact result of an operation, but rather that it can be used to compute exact *bounds* for the result of an operation or a whole sequence of operations. Obtaining such mathematically safe bounds for results is achieved by the proper use of suitable directed roundings. For example, refer to Kulisch, [210], and Kulisch/Miranker, [211], for such a mathematically sound basis of computer arithmetic.

Another mathematical subject which may be treated independently of computers (but reveals its true power only when used on computers as we will see shortly) is interval arithmetic. This kind of arithmetic can be interpreted as a calculus of inequalities – thus formalizing what every analyst is doing with pencil and paper. For example, interval addition $[a, b]+[c, d] = [a+c, b+d]$ is just a compact notation of the argument "if $a \leq x \leq b$ and $c \leq y \leq d$ then $a+c \leq x+y \leq b+d$". Similarly, the interval evaluation of an elementary function, the natural logarithm, say, $ln([a, b]) = [ln(a), ln(b)]$ is just a short notation for "if $a \leq x \leq b$ then $ln(a) \leq ln(x) \leq ln(b)$". The great advantage of such a *calculus of inequalities* is quite obvious, since now it is easy to compute bounds for functions defined by more complicated expressions. This fundamental property of interval arithmetic is also called *inclusion monotonicity.* For details on interval arithmetic, see [247], [248], [13], [256].

Applying interval arithmetic with pencil and paper only would certainly be a very unpleasent and error-prone task. However, in connection with the previously discussed subject, namely floating-point arithmetic with suitably defined roundings, the basic objective of interval arithmetic can still be maintained in a mathematically rigorous way: to obtain true bounds for the result of each operation. On a computer these bounds will of course always be floating-point numbers. For example, if $\bigtriangledown r$ means the largest floating-point number less than or equal to r (rounding downwards) and $\bigtriangleup r$ means the smallest floating-point number greater than or equal to r (rounding upwards) then the inequality $a \leq x \leq b$ can be represented on the computer as $x \in [\bigtriangledown a, \bigtriangleup b]$. This interval is a slight coarsening of the original inequality, at least if $\bigtriangleup b \neq b$ or $\bigtriangledown a \neq a$. We say the real interval $[a, b]$ has been rounded outwards to the smallest interval $[\bigtriangledown a, \bigtriangleup b]$ whose bounds are floating-point numbers and which contains $[a, b]$.

Now each single operation on the computer has to be realized with the same philosophy: instead of computing the exact result, we compute an interval with floating-point bounds containing the exact result. We call such an arithmetic a *machine interval arithmetic* – obviously the concept of inclusion monotonicity carries over to a machine interval arithmetic. Thus, using machine interval arithmetic we are able to deduce *numerically on the computer* properties and results for a continuous set of numbers (e.g. intervals) and consequently we are also able to perform mathematical operations on and make mathematical statements about continuous sets in other mathematical spaces like $I\!R^n$ and even in function spaces.

This is the elementary but extremely important basis on which in the last two to three decades many numerical methods have been developed which by use of interval analysis obtain results containing the true solution of the underlying problems. Results in this sense are always intervals (in a suitable space) which are mathematically verified to contain the unknown solution. We call such methods *self-validating methods* or *methods with result verification.*

This verification is a direct consequence of inclusion monotonicity in the case where the problem can be solved by just a sequence of operations of interval arithmetic. However, if the problem to be solved contains some equation(s) with one or more unknowns or if we have problems in function spaces containing differentials and/or integrals then these self-validating methods must use some mathematical theorem(s) to verify that the computed intervals contain the solution. Ideally it should be possible to verify the hypothesis of such theorems with the help of the computer itself. In this case the computation of the enclosing intervals and the verification that they contain a solution can both be carried out on the computer.

One typical class of theorems which can be checked on a computer are fixed-point theorems which require that a function f maps some set U into itself: $f(U) \subseteq U$. Such a condition can be verified on a computer in the following way. Compute an interval $[U]$ which contains the set U. Next compute an interval evaluation $f([U])$, i.e. compute an interval $[V]$ containing all possible values $f(u)$ for $u \in [U]$. Very often obtaining such an interval evaluation of f over $[U]$ is straightforward. Finally we check whether $[V]$ is contained in U. If this is the case then we have $f(U) \subseteq f([U]) \subseteq [V] \subseteq U$ which proves that the hypotheses $f(U) \subseteq U$ is satisfied and this can be checked completely on the computer.

If there are additional hypotheses in the theorem it may perhaps also be possible to check them on the computer or, if this is not the case, the self-validating method must be designed in such a way that it admits only functions f which satisfy the additional hypotheses. E.g. if we use the Banach fixed-point theorem and we do not have a method to compute an upper bound for the Lipschitz-constant for f then we have to design our algorithm such that only functions f with Lipschitz-constant smaller than 1 are admitted. In the other case also this second hypotheses may be checked by the computer itself. Similarly, when using Brouwer's fixed-point theorem, we must make sure that the self-validating method uses only continuous functions. The computational part of this theorem, $f(U) \subseteq U$, can be tested as described above.

When the hypotheses of such a theorem are verified to be true by use of a computer then of course the assertions of the theorem are true also and in the case of a fixed-point theorem this means that a fixed-point exists, i.e. that $u = f(u)$ has a solution (which might even be unique, depending on which theorem precisely we are using). In this case it is no exaggeration to state that the existence of a solution has been verified by the computer since the most essential part of the work was carried out by interval computations (namely to show that $f(U) \subseteq U$).

Other classes of theorems can also be accessible for interval computations. E.g. theorems which provide bounds for eigenvalue problems need not be (and usually are not) of fixed-point type. Nevertheless, many of them can be used to compute rigorous bounds for eigenvalues. One simple example here is the Gerschgorin circle theorem (see e.g. [309] and Section 3.3).

Writing software which implements such methods is usually much more difficult than writing software in classical numerics where only approximations are computed. Since the result of the program should have mathematical rigour it is necessary that the complete program be designed just as if it were a mathematical proof. One single roundoff error in the wrong place might destroy the whole result completely!

Another difficulty is that the common classical programming languages are in general unsuited for writing software like this since they very often lack of the necessary clean floating-point arithmetic and also of many other features which we will shortly discuss in the next subsection.

The purpose of this treatise therefore is to demonstrate to the expert and the almost-expert how such self-verifying methods can be implemented and to furnish the users which are less interested in the theory but still want to apply these methods with programs that can be used like a traditional software library, but have much stronger results: when they compute a result, this result is verified to be correct. If no result is produced, the program says that it cannot solve the problem within the given limits of resources (time, precision, memory, etc.).

In addition to the verification aspect our methods and programs often achieve highly accurate results by using an additional operation, the so-called *exact scalar product*, which is not present in classical computer arithmetic and programming languages. In [210] and [211] it is shown that with this additional operation the computer arithmetic can be made optimal in many mathematical product spaces. However, the use of this new operation, which is implemented by supplying a data type `dotprecision` in PASCAL-XSC, representing a long accumulator, is also unusual for many programmers. Therefore many methods and programs in this treatise are also intended to demonstrate the efficient use of this new operation and data type and to show how it can be used to achieve flexibility in the computational precision although only one real format is actually being used.

1.2 On the Programming Language used in this Treatise

Almost all presently available programming languages do not offer a clean floating-point arithmetic in the sense described in the previous section. Even though most computers nowadays are equipped with IEEE arithmetic in hardware, which is a computer arithmetic standard including all features mentioned above except the exact scalar product, the programming languages usually have no or only very limited access to such important features like e.g. rounding control.

There exist also only few programming languages which allow the definition of new operators or which support dynamic arrays. Both of these features are very important for writing mathematical computer programs. They help very much in keeping the program code short and compact which is essential when it comes to checking a program for correctness. User defined operators allow a natural mathematical notation instead of an endless list of subroutine calls, whereas dynamic arrays make it possible to use always exactly the correct array sizes thus being free of caring about unused array elements and passing array dimensions as parameters to many procedures and functions. Programs using such features are far easier to write, to maintain and to check for correctness.

Another aspect from the mathematical point of view is that a compiler should be very serious about checking as many error conditions as possible in order to avoid any unintended programming errors both during compile time and during runtime of the programs. Many systems e.g. do not allow index checking for arrays or even overflow detection of integer arithmetic. Not being notified about such possible error conditions is a severe lack of security and makes it often very tedious to find programming errors and fixing bugs.

From these aspects the choice of PASCAL-XSC for our programs is nearly the only possibility since with this programming language we have an operator concept, a clean floating-point arithmetic with access to directed roundings, interval, complex and complex interval arithmetic, reliable standard functions, dynamic arrays and many more necessary and useful features.

One especially important feature which is not even present in all members of the -XSC family (PASCAL-, ACRITH-, C- and FORTRAN-XSC) are the scalar product expressions which simplify the use of the exact scalar product to a large extent. Also the availability of the data type `dotprecision` representing a long accumulator distinguishes PASCAL-XSC (and the other -XSC languages) from all other programming languages.

Another aspect is that at least at the moment PASCAL-XSC is the most portable system among the -XSC languages which is also important from the point of view of program correctness, since changes in a program are not necessary as it is ported from one system to another. Thus no errors can be introduced by adapting programs to new platforms.

The complete language definition of PASCAL-XSC is contained in the Language Reference Manual, see [173] and [174].

1.3 On the Problems discussed in this Treatise

We assume that the reader is familiar with the basics of interval arithmetic which can be found in [247], [248], [13], [256], see also [124]. A deeper knowledge of the various topics treated here is not necessary (though sometimes it might help more). For each problem that we treat we start with a discussion of the theoretical background and of the algorithm which we want to implement. This discussion should suffice to understand the method and its implementation. Example programs show how the method should be used in practice.

In Chapter 2 we start with the discussion of systems of linear equations. This chapter is divided into two parts, the first one treating systems with full coefficient matrices and the second part systems with banded coefficient matrix. For full systems we also consider the over- and under-determined cases as well as the problem of computing inverse matrices and pseudo inverse matrices. In all cases we present algorithms which compute verified enclosures for the solutions. These algorithms will be extended to all four basic data types: real, interval, complex and complex interval. The algorithms for full systems are originally designed by Rump, [280], [285], whereas in the second part of this chapter for systems with band matrices we modify this algorithm by introducing a new method for the solution of triangular systems with band matrices.

In Chapter 3 eigenvalue problems are treated. Here again the method for ordinary eigenvalue problems are from Rump, [280], [285], and the method for the general eigenvalue problem is a simple generalization of this method. Whereas this method is iterative and needs approximate eigenvalues and eigenvectors as starting values for the iteration, we do not need such values for the method for the symmetric eigenvalue problem. The method for the symmetric case is from Lohner, [231], and can also be generalized to Hermitian eigenvalue problems. The basic principle of this method is to compute an approximate eigensystem, transforming the original matrix with this system and computing eigenvalue enclosures from the transformed system by use of the Gerschgorin circle theorem.

Automatic differentiation is a relatively new tool in numerical analysis which makes is possible to compute derivative values without truncation errors as e.g. with divided differences. In Chapter 5 we present several variants of automatic differentiation such as the computation of Taylor-coefficients, the computation of gradients using the so-called reverse mode of automatic differentiation which is very fast and a similarly fast method for the computation of directional derivatives. In all cases the use of interval arithmetic permits us even to *enclose* the values of the derivatives.

Chapter 5.3.5 contains the description and implementation of a high precision arithmetic in so-called *staggered correction format*. This is an excellent example for the great flexibility in precision which we can achieve when we make use of the long accumulator despite of the fact that we have only one data format available for real numbers. Of course this staggered format can also be used to represent high precision intervals which makes it an ideal tool for the computation of highly accurate results even for ill-conditioned problems.

In Chapter 10 we present some simple methods for the verified computation of definite integrals. Here we will make use of the automatic differentiation module from Chapter 5 for the computation of Taylor coefficients to compute enclosures for the remainder terms of the numerical integration rules. We discuss the trapezoidal rule, Simpson's rule and Romberg integration. For each of these rules explicit error terms are known which contain higher derivatives of the integrand evaluated at unkown intermediate points. It turns out that the combination of interval arithmetic and automatic differentiation is an exceptionally well suited tool in this case, since an interval can be used to enclose the unknown intermediate argument and automatic differentiation can be used to compute the high derivative (i.e. the Taylor coefficients) over this interval thus obtaining a rigorous estimate of the error term. For the trapezoidal and Simpson's rule we will use fixed step sizes only, however, for Romberg integration we will develop a simple adaptive step size control which is not intended to be optimal in any sense but which still is powerful enough to obtain satisfying results for quite difficult integrals.

Chapter 2

Systems of Linear Equations

One of the most frequent tasks in Numerical Analysis is the solution of Systems of linear equations

$$Ax = b \tag{2.1}$$

with an $n \times n$ matrix A and a right hand side $b \in \mathbb{R}^n$. Many different numerical algorithms contain this task as a subproblem.

As a generalization to this problem with a square matrix we often encounter *over-* or *under-determined* systems, i.e. systems where A is not a square, but rather an $m \times n$ matrix with $m > n$ in the over- and $m < n$ in the under-determined case. In the over-determined case a vector $x \in \mathbb{R}^n$ is sought whose residuum $b - Ax$ has minimal Euclidian norm whereas in the under-determined case a solution $x \in \mathbb{R}^n$ of (2.1) is sought which has minimal norm.

Also the inversion of the matrix A is a problem of this type. Here, the right hand side of (2.1) has to be replaced by the $n \times n$ identity matrix I and the solution X, which is a matrix now, will be the inverse A^{-1} of A. In the over- or under-determined case this matrix X will be the Moore-Penrose pseudo inverse A^+ of A (if A has full rank).

There are numerous methods and algorithms which compute approximations to the solution x in floating-point arithmetic. However, usually it is not clear how good these approximations are, or if there exists a unique solution at all. In general, it is not possible to answer these questions with mathematical rigour if only floating-point approximations are used. These problems become especially difficult if the matrix A is ill conditioned.

In this Chapter we will present some algorithms which answer the questions about existence and accuracy automatically once their execution is completed successfully. Even very ill conditioned problems can be solved with these algorithms. Most of the algorithms presented here can be found in [285]. All algorithms work for all four basic numerical PASCAL-XSC data types: *real, interval, complex* and *complex interval.*

We divide this Chapter into two parts, the first part deals with systems whose coefficient matrix is dense, whereas the second part treats systems with banded matrices as coefficient matrix.

2.1 Systems with Dense Matrices

2.1.1 Introduction

Here we assume the coefficient matrix A in (2.1) to be dense, i.e. in a PASCAL-XSC program we use a square matrix of type *rmatrix*, *imatrix*, *cmatrix* or *cimatrix* to store A and we do not consider any special structure of the elements of A.

Our goal is to write a PASCAL-XSC program that *verifies the existence* of a solution and *computes an enclosure* for this solution for each of the following types of problems:

(s) compute an enclosure for the solution of system (2.1) for a *square* $n \times n$ matrix A.

(o) compute an enclosure for the solution of system (2.1) in the *over-determined* case, i.e. for an $m \times n$ matrix A where $m > n$.

(u) compute an enclosure for the solution of system (2.1) in the *under-determined* case, i.e. for an $m \times n$ matrix A where $m < n$.

(S) compute an enclosure of the *inverse* A^{-1} of A.

(O) compute an enclosure of the *pseudo inverse* A^{+} of A in the *over-determined* case, i.e. for an $m \times n$ matrix A where $m > n$.

(U) compute an enclosure of the *pseudo inverse* A^{+} of A in the *under-determined* case, i.e. for an $m \times n$ matrix A where $m < n$.

We also want these six problems to be solved for all four basic numerical PASCAL-XSC data typtes: *real, interval, complex* and *complex interval.* In the following Section 2.1.2 we will briefly outline the solution methods. The corresponding algorithmic description can be found in Section 2.1.3 and the PASCAL-XSC program code will be presented in Section 2.1.5.

2.1.2 Theoretical Background

In this section we give only a brief summary of the theory of the enclosure methods for our six problems. A more detailed presentation can be found e.g. in [285].

Starting with problem (s) we first assume that we have an approximate solution $\tilde{x}$ and an approximate inverse R of the square matrix A. Rather than computing an enclosure for the solution directly we will try to enclose the error of the approximate solution which yields much higher accuracy. The error $y = x - \tilde{x}$ of the true solution

x satisfies the equation

$$Ay = b - A\tilde{x} \tag{2.2}$$

which can be multiplied by R and rewritten in the form

$$y = R(b - A\tilde{x}) + (I - RA)y \tag{2.3}$$

or with $f(y) := R(b - A\tilde{x}) + (I - RA)y$

$$y = f(y)\ . \tag{2.4}$$

This is an equation in fixed point form for the error y. If R is a sufficiently good approximation of A^{-1} then an iteration based on (2.4) can be expected to converge since then $I - RA$ will have a small spectral radius.

Therefore, from (2.4) we derive the following iteration, where we already use interval arithmetic and intervals $[y]_k$ for y:

$$[y]_{k+1} = R\Diamond(b - A\tilde{x}) + \Diamond(I - RA)[y]_k \tag{2.5}$$

or

$$[y]_{k+1} = F([y]_k) \tag{2.6}$$

where F is the interval extension of f.

Here $\Diamond$ means that the succeeding operations have to be executed exactly and the result is rounded to an enclosing interval (-vector or -matrix). Since in the computation of the defect $b - A\tilde{x}$ and of the iteration matrix $I - RA$ serious cancellations of leading digits must be expected these should be computed using the exact scalar product, i.e. each component is computed exactly and then rounded to a machine interval. For this purpose the scalar product expressions of PASCAL-XSC will be used extensively in the implementations. With $z := R\Diamond(b - A\tilde{x})$ and $C := \Diamond(I - RA)$ now (2.5) can be written as:

$$[y]_{k+1} = z + C[y]_k\ .$$

In order to prove the existence of a solution of (2.2) and thus of (2.1), we use a fixed point theorem, i.e. Brouwer's fixed point theorem, which applies as soon as we have at some iteration index $k+1$ an inclusion property of the form

$$[y]_{k+1} = F([y]_k) \subset [y]_k^\circ \tag{2.7}$$

where $[y]_k^\circ$ means the interior of $[y]_k$. If this *inclusion test* (2.7) holds, then the iteration function f maps $[y]_k$ into itself and from Brouwers fixed point theorem it follows that f has a fixed point y^* which is contained in $[y]_k$ and thus even in $[y]_{k+1}$. The requirement that $[y]_k$ is mapped into its interior, makes shure, in addition, that this fixed point is also unique, i.e. (2.2) has a unique solution y^* and thus (2.1) also has a unique solution $x^* = \tilde{x} + y^*$.

Remark:

According to [285] the inclusion test (2.7), when satisfied, implies that the spectral radius of C (and even that of $|C|$, which is the matrix of absolute values of C) is less then 1 which ensures the convergence of the iteration (also in the interval case). Furthermore, this implies also the nonsingularity of R and of A and thus the uniqueness of the fixed point.

A problem which still remains is, that we do not know if we can succeed in achieving condition (2.7). E.g. in the trivial case of $n = 1$ with $a_{11} = 0.1, b = 0$ and $\tilde{x} = 1$ the iteration will converge to the unique solution $x^* = 0$ but will do this monotonically decreasing and (2.7) will never be satisfied.

To force (2.7) we therefore introduce the so called ϵ-*inflation* which blows up the intervals somewhat, in order to "catch" a nearby fixed point. For a real interval $[w]$ we denote this with the functional notation $\text{blow}([w], \epsilon)$ where

$$\text{blow}([w], \epsilon) := \begin{cases} (1+\epsilon)[w] - \epsilon[w] & \text{, if } \text{diam}([w]) > 0, \\ [\text{pred}(w), \text{succ}(w)] & \text{, if } \text{diam}([w]) = 0 \end{cases} \tag{2.8}$$

where $\text{diam}([w])$ is the diameter of the interval ($\text{diam}([w]) = \overline{w} - \underline{w}$) and in the case $\text{diam}([w]) = 0$, $[w] = w$ is assumed to be a floating point number and $\text{pred}(w)$, $\text{succ}(w)$ are its predecessor and successor in the floating point screen. Similarly, we use the ϵ-inflation $\text{blow}(\cdot)$ also for interval vectors and matrices, where it is applied componentwise.

It can be shown, e.g. [289] that (2.7) will always be satisfied after a finite number of iteration steps, whenever the absolute value $|C|$ of iteration matrix C has spectral radius less than 1.

Up to this point we did not yet say how we compute our approximate solution $\tilde{x}$ and the approximate inverse R. In principle there is no special requirement about these quantities, we could even just guess them. However, the results of the enclosure algorithm will of course depend on the quality of the approximations.

We now sketch the method we use in our PASCAL-XSC program for the computation of R and $\tilde{x}$.

To begin with, we do not use a special algorithm for the computation of the approximate solution, since we must compute an approximate inverse $R \approx A^{-1}$ anyway. Thus we also have immediately an approximate solution $\tilde{x} := Rb$.

However, the quality of this approximation is often not sufficient for the interval iteration to converge fast. Therefore, we first improve this approximation by use of an iterated defect correction:

$$\tilde{x}_{k+1} = \tilde{x}_k + R(b - A\tilde{x}_k) \tag{2.9}$$

using floating point arithmetic only and the exact scalar product for the defect $b - A\tilde{x}_k$. Improving the approximation in floating point first is much cheaper than computing many interval iterations later.

For the computation of the approximate inverse R we use the Gauss-Jordan algorithm with column pivoting, which can be found in many numerical text books, e.g. [309]. We do not repeat the algorithm here since it is very standard (and it is listed as PASCAL-XSC code in Section 2.1.5 anyway). However, we explain a minor change in the algorithm which was included in order to make the algorithm more robust in the case of almost singular matrices. This modification concerns the elimination steps of the Gauss-Jordan algorithm which may produce exact zeros on the computer because of rounding errors. Later in the computation, when a pivot element is to be searched for, it could be the case that no pivot can be found since all candidates are exact zeros on the machine, thus causing a breakdown of the Gauss-Jordan algorithm. Therefore we replace any exact zero value which was produced in an elimination step by a nonzero value in the order of magnitude of the roundoff error, i.e. we replace the result of $a - a = 0$ by ϵa where ϵ is the relative machine precision. This modification forced the Gauss-Jordan algorithm to execute completely for any non singular matrix (and also many singular matrices) and, additionally, now the failure of finding a pivot element also means that the zeros have been already in the input matrix, which therefore must surely be singular.

In very ill conditioned cases the quality of R computed this way will not be sufficient, i.e. the spectral radius of $|C|$ will not be less then 1 and therefore the inclusion test will never be satisfied. In this case we stop the interval iterations after a maximal number of iterations (10, say) and recompute the approximate inverse in higher precision. The method how we do this goes back to Rump, [280], why we will call it Rump's device.

Assume we have an approximate inverse R (as we do from the Gauss-Jordan algorithm), then even if A is very ill conditioned, the matrix RA is ususally very much better conditioned than A is. Now the simple relation

$$A^{-1} = (RA)^{-1}R \tag{2.10}$$

suggests that we compute another approximate inverse S of RA and take the product of S and R as a better approximation of A^{-1}. Since we can compute this product exactly with the aid of the exact scalar product in the long accumulator, it is also easy to approximate it by the sum of two matrices R_1 and R_2.

Summarizing, we can compute an approximate inverse $R_1 + R_2$ of double length (stored in two real floating point matrices R_1, R_2) by the following steps:

1. compute an approximate inverse R of A with Gauss-Jordan.
2. compute RA.
3. compute an approximate inverse S of RA with Gauss-Jordan.
4. compute SR in the long accumulator, and store it as sum of two floating point matrices R_1 and R_2.

Now that we have an approximate inverse $R := R_1 + R_2$ of double length, we can execute the algorithm described above with this R by using the exact scalar product whenever $R = R_1 + R_2$ is used in the algorithm.

The overall strategy of our method finally can be summarized in two steps:

Part I compute an approximate inverse R of single length and execute the enclosure algorithm. If this fails then goto

Part II improve the approximate inverse by Rump's device and execute the enclosure algorithm with the double length approximate inverse $R = R_1 + R_2$.

Now that we have treated the case of one linear equation with one right hand side b we turn to the inversion of a matrix A. The inverse $X = A^{-1}$ is obviously the solution of the matrix-equation $AX = I$ with the $n \times n$ identity matrix on the right hand side. This equation can of course be solved column wise, thus obtaining the columns of $X = A^{-1}$ successively.

For the n individual matrix-vector equations which have to be solved we use the algorithm just presented. In order to keep the computational overhead small we can make use of some simple observations which help us to avoid the recomputation of intermediate quantities in the algorithm: Since all n equations have the same coefficient matrix A it would be a big waste of computation time to recompute the approximate inverse R and the iteration matrix $C = \Diamond(I - RA)$ for each of the n matrix-vector equations.

Rather, we will implement the algorithm for the square case (s) in such a way, that the approximate inverse R (or R_1, R_2 if a double length inverse was needed) as well as its residual matrix $C = \Diamond(I - RA)$ (or $C = \Diamond(I - R_1A - R_2A)$) will be saved after the first computation and will be reused in the solution of the following equations.

In the PASCAL-XSC code the procedure which computes solutions of the square matrix-vector equation (i.e. the local procedure `LSS`) must communicate with its calling procedures for matrix-equations which quantities of the algorithm have already been computed (R, R_1, R_2, $\Diamond(I - RA)$ and $\Diamond(I - R_1A - R_2A)$) and can be reused in a future call. For this communication a flag (named `FLAGS`) is used in the PASCAL-XSC program code in Section 2.1.5 which helps to avoid these unnecessary recomputations.

Next, we consider the cases of over- and of under-determined systems. (For more details on the theory see, e.g. [309]). In both cases we assume the $m \times n$-matrix A to have full rank, i.e. in the over-determined case (case (o), $m \geq n$) A has rank n and in the under-determined case (case (u), $m \leq n$) A has rank m. Here x is an n-vector and b is an m-vector.

In case(o) the system (2.1) has no solution in general. Therefore, we are rather interested in a vector x which minimizes the Euclidian norm of the residual vector $r = b - Ax$, or, equivalently, the square of this norm, i.e. we seek the solution of the linear least squares problem:

$$\|Ax - b\|_2^2 = min \ .$$

It is well known (e.g. [309], [308]) that such an x is uniquely determined (if A has full rank) and that it is the solution of the so called system of *normal equations*

$$A^H Ax = A^H b \tag{2.11}$$

where A^H is the Hermitian matrix of A, i.e. the transposed matrix in the real case.

We could now go ahead, compute $A^H A$ and $A^H b$ and solve the resulting square $n \times n$ system by use of the just previously presented method. However, as is well known, $A^H A$ has usually very bad condition and, moreover, on the computer it can only be obtained with roundoff errors or as an interval matrix which makes the solution of this system difficult.

Instead, we follow the suggestion of [285] and rewrite (2.11) as a larger square $(n+m) \times (n+m)$-system, which can be solved by the previous method to very high accuracy (but also with much higher computational effort if $m >> n$).

Introducing a new m-vector $y = Ax - b$ we immediately obtain $A^H y = 0$ from (2.11). We write these two equations in block form

$$\begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \tag{2.12}$$

which is a square $(n+m) \times (n+m)$ system (I is the $m \times m$ identity matrix here). This system has much better condition than the original normal equations.

Now it is straight forward to solve (2.12) by the method for square systems. The x-part of the resulting enclosure then is an enclosure for the solution x of the normal equations (2.11).

In a very similar way we also proceed in the case of under-determined systems. Here, the system (2.1) usually has infinitely many solutions and we are interested in the vector y among these solutions x which has minimal Euclidian norm. This vector can be determined as $y = A^H x$, where x is the solution of $AA^H x = b$. Again we write these two equations in block form

$$\begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \tag{2.13}$$

which again is a square $(n+m) \times (n+m)$ system (here I is the $n \times n$ identity matrix).

Again, we solve this system (2.13) by use of the method for square systems. The y-part of the resulting enclosure then yields an enclosure for the solution of our problem.

Finally we treat the two cases (O) and (U), i.e. the computation of the Moore-Penrose pseudo inverse A^+ of A in the over- and under-determined cases (here we also assume A to have full rank).

In case (O) the $n \times m$-matrix A^+ ($m \geq n$) is given by (see e.g. [309], [308]).

$$A^+ = (A^H A)^{-1} A^H \tag{2.14}$$

or, equivalently, by the solution $Y = A^+$ of the matrix equation $A^H AY = A^H$. Introducing the $m \times m$-matrix $X = AY - I$ we see that $A^H X = 0$ and, as previously, we can write these two equations in block form:

$$\begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \cdot \begin{pmatrix} Y \\ X \end{pmatrix} = \begin{pmatrix} I \\ 0 \end{pmatrix} \tag{2.15}$$

where both identity matrices are $m \times m$.

Solving this equation column-wise with the algorithm for square matrices we get an enclosure for the pseudo inverse A^+ by extracting the Y-block from the computed enclosure.

Since the pseudo inverse has the property $(A^+)^H = (A^H)^+$ it follows from (2.14) that in case (U) ($m \le n$) we have

$$A^+ = A^H(AA^H)^{-1}$$

or, equivalently, A^+ is the solution $Y = A^+$ of $YAA^H = A^H$. With the $n \times n$-matrix $X = YA - I$ we see that $XA^H = 0$. Taking the hermitian in the two latter equations we can again write these in a block form:

$$\begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \cdot \begin{pmatrix} Y^H \\ X^H \end{pmatrix} = \begin{pmatrix} I \\ 0 \end{pmatrix} \tag{2.16}$$

with the identity matrices being $n \times n$.

Solving this equation column-wise with the algorithm for square matrices we get an enclosure for the pseudo inverse A^+ by extracting the Y^H-block from the computed enclosure and taking its hermitian matrix as the result.

Remark:

We should stress that the alogrithms which were presented here for the cases (o), (u), (O) and (U) generally deliver very narrow bounds for the solution, but that they are computationally very expensive, at least in the case where $|m - n|$ is large since the solution is always computed by use of a system of dimension $n + m$. There are other methods which can enclose the solution more efficiently (though perhaps less accurate), see Section 2.1.7. The reasons for the presentation of our methods are their high accuracy and that they are straight forward and can be implemented in PASCAL-XSC in some 10-20 lines of code.

According to a remark in [285] the number of operations can be reduced substantially in our algorithms for the cases (o), (u), (O) and (U) if we carefully observe the special structure of the block coeffient matrices B and compute only those elements of the approximate inverse R and its residual matrix $I - RB$ which are actually needed.

2.1.3 Algorithms

According to Section 2.1.2 we can outline the following algorithms in a pseudo Pascal notation.

Algorithm 2.1: Solution of problem (s): {procedure}

{Compute enclosure for solution of square linear System $Ax = b$}

Part I (approximate inverse of single length)

I.1 compute an approximate inverse R of A (e.g. using Gauss-Jordan algorithm)

I.2 compute an approximation $\tilde{x} := Rb$ of x
improve $\tilde{x}$ by an iterated defect correction :
repeat
$\tilde{x} := \tilde{x} + R(b - A\tilde{x})$
until $\tilde{x}$ accurate enough or max. iteration count exceeded

I.3 compute enclosures for the residuum:
$Z := R\Diamond(b - A\tilde{x})$
and for the iteration matrix:
$C := \Diamond(I - RA)$

I.4 interval iteration
$Y := Z$
repeat
$Y_A := \text{blow}(Y, \epsilon)$ {ϵ - inflation }
$Y := Z + C \cdot Y_A$
until $Y \subset \text{int}(Y_A)$ or max. iteration count exceeded

I.5 **if** $Y \subset \text{int}(Y_A)$ **then**
a unique solution x exists and $x \in \tilde{x} + Y$
else
if in Part I **then**
Part I failed, goto Part II with $R_1 := R$
else
algorithm failed, matrix A ill conditioned or singular

Part II (approximate inverse of double length)

II.1 (compute an approximate inverse $R := R_1 + R_2$ of A as follows:)
$S := R_1 \cdot A$
compute an approximate inverse S_1 for S (e.g. using Gauss-Jordan algorithm)
$S := S_1 \cdot R_1$
$R_2 := S_1 \cdot R_1 - S$
$R_1 := S$

II.2 goto step I.2 of Part I.

Algorithm 2.2: Solution of problem (o): {procedure}

{Compute enclosure for solution of over-determined linear System $Ax = b$}

1. $A_{big} := \begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \in I\!R^{(n+m)\times(n+m)}$

$B_{big} := \begin{pmatrix} b \\ 0 \end{pmatrix} \in I\!R^{n+m}$

$Y_{big} := \begin{pmatrix} x \\ y \end{pmatrix} \in I\!R^{n+m}$

2. solve $A_{big}Y_{big} = B_{big}$ by use of Algorithm 2.2.1
3. block x from the vector Y_{big} is the wanted enclosure.

Algorithm 2.3: Solution of problem (u): {procedure}

{Compute enclosure for solution of under-determined linear System $Ax = b$}

1. $A_{big} := \begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \in I\!R^{(n+m)\times(n+m)}$

   $B_{big} := \begin{pmatrix} 0 \\ b \end{pmatrix} \in I\!R^{n+m}$

   $Y_{big} := \begin{pmatrix} x \\ y \end{pmatrix} \in I\!R^{n+m}$
2. solve $A_{big}Y_{big} = B_{big}$ by use of Algorithm 2.2.1
3. block x from the vector Y_{big} is the wanted enclosure.

Algorithm 2.4: Solution of problem (S): {procedure}

{Compute enclosure for the inverse of the square matrix A}

1. (solve $AX = I$ column wise)
   for $i := 1$ **to** n **do**
   begin
   $b_i := e_i$ ($= i$-th unit vector)
   solve $Ax_i = b_i$ by use of Algorithm 2.2.1
   end
2. $X = (x_1, \ldots, x_n)$ is the wanted enclosure.

Algorithm 2.5: Solution of problem (O): {procedure}

{Compute enclosure for the pseudo inverse of the $m \times n$ matrix $A, m > n$}

1. $A_{big} := \begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \in I\!R^{(n+m)\times(n+m)}$, $I \in I\!R^{m\times m}$

   $B_{big} := \begin{pmatrix} I \\ 0 \end{pmatrix} \in I\!R^{n+m\times m}$, $I \in I\!R^{m\times m}, 0 \in I\!R^{n\times m}$

   $Y_{big} := \begin{pmatrix} X \\ Y \end{pmatrix} \in I\!R^{n+m\times m}$, $X \in I\!R^{m\times m}, Y \in I\!R^{n\times m}$
2. solve $A_{big}Y_{big} = B_{big}$ by use of Algorithm 2.2.4
3. block X from the matrix Y_{big} is the wanted enclosure.

Algorithm 2.6: Solution of problem (U): {procedure}

{Compute enclosure for the pseudo inverse of the $m \times n$ matrix $A, m < n$}

1. $A_{big} := \begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \in I\!R^{(n+m)\times(n+m)},\ I \in I\!R^{m\times m}$

   $B_{big} := \begin{pmatrix} I \\ 0 \end{pmatrix} \in I\!R^{n+m\times n},\ I \in I\!R^{n\times n}, 0 \in I\!R^{m\times n}$

   $Y_{big} := \begin{pmatrix} Y^H \\ X^H \end{pmatrix} \in I\!R^{n+m\times n},\ X \in I\!R^{n\times n}, Y \in I\!R^{n\times m}$
2. solve $A_{big}Y_{big} = B_{big}$ by use of Algorithm 2.2.4
3. block X from the matrix Y_{big} is the wanted enclosure.

2.1.4 Applicability of the Algorithms

The algorithms presented in Section 2.1.4 can be applied to any system of linear equations which can be stored in the floating point system on the computer. They will, in general, succeed in finding and enclosing a solution or, if they do not succeed will tell the user so. In the latter case the user will know that the problem is very ill conditioned or that the matrix A is singular.

In the following implementation in PASCAL-XSC there is the chance that if the input data contains large numbers or if the inverse of A or the solution itself contain large numbers there may occur an overflow in which case the algorithms break down uncontrolled. In practical applications this has never been observed, however. This could also be avoided by including the floating point exception handling which PASCAL-XSC offers for IEEE floating point arithmetic.

If the problem contains interval data in the matrix A or in the right hand side b then the algorithms can be applied, too, however, the exact solution set is usually overestimated by the enclosure which is computed by the algorithm. Usually this overestimation becomes larger as the diameter of the input interval increases. Also for certain classes of matrices for which the optimal interval hull of the solution set can be computed (e.g. interval M-matrices) this algorithm will usually overestimate the solution set. This overestimation can be estimated by use of certain quantities appearing in the algorithm. We will not discuss this here, however. For details see e.g. [288] and [91].

2.1.5 PASCAL-XSC Program Code

The PASCAL-XSC program which is given here is written for the case of real input data, i.e. A is of type *rmatrix* and b is of type *rvector*. However, because of the very clear structure of PASCAL-XSC it is easy to transform this program code to make use of the other data types *interval, complex* and *complex interval*. The changes in the program are mainly changes of the data type of certain variables and functions. We will indicate the necessary changes at the end of this section.

The following module `lss_aprx` contains the procedure `MINV` which computes an approximate inverse of the input matrix `A` of type *rmatrix* by use of the Gauss-Jordan algorithm (see e.g. [309]). The parameter `A` is for input and output. The second parameter `err` indicates if an approximate inverse could be computed (`err=0`) or if the computation failed (`err=1`).

There is, however, one trick included in this algorithm: in order to be able to complete the algorithm even for ill conditioned matrices we avoid to compute matrix elements which are exactly zero in the elimination step. For ill conditioned matrices these elements could appear later as pivot elements, thus forcing the computations to break down. If the elimination step would yield an element `a[l,k]` which is rounded to an exact zero we rather replace it by `eps*a[l,k]` with the old value of `a[l,k]` and an `eps` close to the relative machine precision. If we then still find a zero pivot element, then it must have been in the input matrix already which means that the input matrix was actually singular. (This is, however, only almost true, since we did not consider the possibility of underflow in the multiplication `eps*a[l,k]`).

The reason why the array index computations look somewhat complicated is that we do not assume that the matrix `A` has the same index ranges in both dimensions, it only has to be square. E.g. if `A` is 10×10 the first index migth run from 0 to 9 whereas the second index could run from -5 to 4.

```
MODULE lss_aprx;

USE mv_ari;

CONST eps = 1E-15;

GLOBAL PROCEDURE MINV( VAR A : RMATRIX; VAR err : INTEGER );
{ MINV computes approximate inverse of the matrix A by use of the }
{ Gauss-Jordan algorithm.                                         }
{ A   : input matrix and output of approximate inverse            }
{ err : Error indicator = 0 everything ok, = 1 matrix singular (?)}

VAR v      : DYNAMIC ARRAY [LB(A,1)..UB(A,1)] OF INTEGER;
    r      : RVECTOR[LB(A,1)..UB(A,1)];
    h      : REAL;
    ok     : BOOLEAN;
    i,j,k,l,
    au1,ao1,au2,ao2 : INTEGER;
BEGIN
   au1:= LB(A,1);  ao1:= UB(A,1);
   au2:= LB(A,2);  ao2:= UB(A,2);

   ok:= ao1-au1 = ao2-au2;   { square matrix ? }

   FOR i:= au1 TO ao1 DO v[i]:= i;

   i := au1-1; { row    index }
   j := au2-1; { column index }

   WHILE (j<ao2) AND ok DO
   BEGIN
      i:= succ(i);  j:= succ(j);  { i and j run simultaniously }
      IF j<ao2 THEN
      BEGIN
         { pivot search : }
         l:= i;
```

```
            FOR k:= i+1 TO ao1 DO  IF ABS(a[k,j]) > ABS(a[l,j]) THEN l:= k;
            { row interchange : }
            IF i<>l THEN
            BEGIN
               r:= a[l];  a[l]:= a[i];  a[i]:= r;
               k:= v[l];  v[l]:= v[i];  v[i]:= k;
            END;
         END;
         { transformation : }
         IF a[i,j]=0.0 THEN ok:= FALSE ELSE
         BEGIN
            h      := 1.0 / a[i,j];
            a[*,j]:= h*RVECTOR(a[*,j]);
            a[i,j]:= h;
            FOR k:= au2 TO ao2 DO
                IF k<>j THEN
                BEGIN
                   FOR l:= au1 TO ao1 DO
                       IF l<>i THEN
                       BEGIN
                          h  := a[l,k] - a[l,j]*a[i,k];
                          IF h=0.0 THEN a[l,k]:= eps*a[l,k]
                                   ELSE a[l,k]:= h;
                       END;
                   a[i,k]:= -a[i,k]*a[i,j];
                END;
         END;
      END;
      { column interchange : }
      IF ok THEN
      FOR k:= au1 TO ao1 DO
      BEGIN
         l:= au2+k-au1;
         WHILE v[k]<>k DO
         BEGIN
            i:= v[k];              v[k]   := v[i];    v[i]   := i;
            j:= au2+i-au1;
            r:= RVECTOR(a[*,j]); a[*,j]:= a[*,l]; a[*,l]:= r;
         END;
      END;
      err:= ORD( NOT ok );
   END {*** MINV ***};

   BEGIN
      { module body empty }
   END.
```

The following module `lss.p` contains the functions and procedures which solve the problems (s), (o), (u), (S), (O) and (U) stated in Section 2.1.1 for a coefficient matrix of type *rmatrix* and a right hand side of type *rvector*.

There are only two global procedures contained in `lss` – `LSS` and `INV` – which can be called from other modules than `lss`. `LSS` accepts linear systems of the form (2.1) with any matrix, checks the dimensions of the coefficient matrix `A` and calls other procedures which handle the square or rectangular cases. Similarly, `INV` checks its parameter `A` and calls other routines for the square or rectangular cases.

The procedures and functions have the following meanings (in order of their static appearance) :

REL:

Computes the maximum relative error of the components of the two parameter vectors `A` and `B`. It is used in the stopping criterion of the real defect iteration for an improvement of the real approximations in the procedures `USE_SINGLE_R` and `USE_DOUBLE_R` (both local to the first procedure `LSS`).

`TOO_BAD`:

Checks the accuracy of the parameter `A`. If at least one component is larger than $[-10^{20}, 10^{20}]$ then the function yields `TRUE` else `FALSE`. It is used to abort the interval iteration if it seems to diverge (again in `USE_SINGLE_R` and `USE_DOUBLE_R`).

`GUESS_ZEROES`:

Tries to guess if a component of the solution is equal to zero. This is assumed to be the case if this component (i) has changed sign in the last two successive iterations or if (ii) it has decreased by more than a factor of 10^{-6} from the previous iterate to the last iterate and if it is less in magnitude than 10^{-6} times the maximum magnitude of all other components of the last iterate. Replacing such components by zero often improves the enclosures by many orders of magnitudes. On the other hand, if the guess was wrong, then zero is still a good approximation such that the enclosures will stay good.

`LSS`:

This is the central procedure of the module. It implements the algorithm for a square system $Ax = b$ by first trying an iteration with an approximate inverse of single length (procedure `USE_SINGLE_R`) and in case of no success with an approximate inverse of double length (procedure `USE_DOUBLE_R`). Input parameters are `A` and `b`. Output parameters are `Y` and `errcode`. The other parameters are for input as well as for output. Since `LSS` is called by the procedures for matrix inversion several times, the computation of the approximate inverses (`R1,R2`) and the iteration matrix `C` is controlled by the parameter `FLAGS` which indicates in successive calls to `LSS` which of the matrices `R1,R2,C` have already been computed in order to avoid unnecessary and costly recomputation of these quantities. For details see the comment at the beginning of `LSS`.

`USE_SINGLE_R`: (local to `LSS`)

This procedure implements the algorithm by using an approximate inverse `R1` of single length which is computed by the procedure `MINV`.

After the computation of `R1` we compute an approximate solution `x1:= R1*b` and execute a defect iteration in floating point arithmetic in order to improve this approximation `x1`. It is very essential here to use scalar product expressions since the computation of the residuum `b-A*x0` must necessarily suffer from severe cancellation. This iteration is stopped if (i) the maximal relative error `p` is less than `delta` $(= 10^{-15})$ or if the relativ error decreases very slowly only, such that `p` becomes larger than `bound` which is multiplied by a factor of $\approx \sqrt{0.1}$ in each iteration starting after the 5th iteration. This means roughly that after the 5th iteration we expect the approximation to gain at least one decimal digit each two iterations, otherwise we stop the iteration process.

Subsequent to this floating point iteration we try to guess if some of the solution components are exactly zero by calling procedure `GUESS_ZEROES`.

Next, we compute an enclosure of the residuum `b-A*x1` in double length (real vector `y0` plus interval vector `Y1`) in order to get still high accuracy for the product with `R1`. Here the repeated computation of `A*x1` inside the `#`-expressions could be avoided if these expressions were computed component wise by use of a variable of type *dotprecision*. However, since these computations are cheap as compared to the rest of the algorithm we prefer this version which is much easier to read.

This residuum is checked if it is exactly the zero vector: in this case `x1` is an exact solution of the system. Thus we can stop the algorithm at this point and need not continue with the costly interval iteration (However, stopping here means that we have an exact solution, but that we do not know if it is unique. If we need uniqueness we have to continue with the interval iteration.)

Finally the iteration matrix `C` is computed (again with scalar product expression because of cancellations) and the interval iteration for the defect `Y1` of the approximation `x1` is executed. Beginning with the 5th iteration the inflation parameter `eps` is multiplied by 5 in each iteration in order to accelerate convergence or divergence. This iteration is stopped (i) if it was successful, i.e. if an iterate `Y1` is contained in its predecessor `YA`, (ii) after a maximum iteration count of 10, or (iii) if the accuracy becomes too bad (indicating divergence).

In the case of a successful interval iteration the resulting enclosure `x1+Y1` of the solution is returned in `Y` and the error code is set to zero.

`USE_DOUBE_R`: (local to `LSS`)

This procedure implements the algorithm by using an approximate inverse `R1+R2` of double length (i.e. `R1` and `R2` are of type *rmatrix* each), which is computed by the use of Rump's device as explained in Section 2.1.2.

The following steps are completely analogous to those in `USE_SINGLE_R` except that wherever the approximate inverse appears the double length representation has to be used. Note that in the floating point defect iteration the iterates are computed in double length also (`x1+x0`) but subsequently only the high value part `x1` is used. Also, note that `y0` is an auxiliary variable in the floating point iteration.

Here again we make use of a repeated computation of `A*x1` only to make the program easier to read. Similarly, the two assignments `D := R2*R1;` and `R2:= #*( R2*R1 - D );` near the beginning of `USE_DOUBE_R` cause repeated computation of `R2*R1`. However, the costs for this are additional n^3 operations. If we do not want this higher overhead, then we have to replace these two assignments by the following double loop:

```
FOR i:= LB(A) to UB(A) do
FOR j:= LB(A) to UB(A) do
BEGIN
   dot    := #( R2[i]*RVECTOR(R1[*,j]) );
   D[i,j] := #*( dot );
   R2[i,j]:= #*( dot - D[i,j] );
```

```
END;
```

Furthermore, in this case we have to add the declarations

```
VAR dot : DOTPRECISION;
    i,j : INTEGER;
```

in the declaration part of USE_DOUBE_R.

SQUARE_LSS:
This procedure treats the case of a square coefficient matrix. The trivial case of a 1×1 matrix is solved explicitly, whereas for $n > 1$ the storage for R1,R2,C and FLAGS is used to call the procedure LSS described above.

OVER_LSS, UNDER_LSS:
These procedure handles the over- and under-determined case respectively of a linear system. Both procedures allocate variables BIG_A,BIG_B and BIG_Y for the augmented problems and initialize BIG_A and BIG_B appropriately according to Section 2.1.2. Then this augmented square system is solved by a call to procedure SQUARE_LSS. Finally those components of the big solution vector are extracted which are needed for the solution of the original problem.

LSS:
This procedure is the only global entry for the solution of linear systems in this module lss. It first checks the parameter for consistent dimensions, then decides which case (s), (o) or (u) has to be treated and calls the appropriate procedure SQUARE_LSS, OVER_LSS or UNDER_LSS.

SQUARE_INV, OVER_INV, UNDER_INV, INV:
These procedures compute enclosures for the inverse matrices (or the pseudo inverse) and are completely analogous to those for linear systems. As mentioned above, INV is the only global procedure for this purpose and calls the other ones appropriately. The only difference to the corresponding -LSS procedures is that they compute the inverse column wise and therefore have to call the local linear system solving procedure LSS several times. Therefore, the matrices R1,R2 and C are defined locally in these procedures and their computation is controlled by the variable FLAG as discussed previously (see local procedure LSS). Note that in procedure SQUARE_INV the resulting columns of the inverse matrix Y are first written into the rows of the result matrix. Therefore, Y has to be transposed at the end of the procedure.

```
MODULE lss;

use  i_ari,mv_ari,mvi_ari,lss_aprx;

CONST zerotest = 1E6;
      delta    = 1E-15;
      sqrt_01  = 0.31622777;
      abort    = 1E20;
```

```
VAR   m,n,dim : INTEGER;


FUNCTION REL ( VAR A,B : RVECTOR ) : REAL;
{ computes componentwise the maximum relative error of A w.r.t B.  }
{ If A[i] and B[i] do not have the same sign or if B[i] = 0, then  }
{ rel. error := 0 for this component.                              }
{ A is always the new value of an iteration, B the old one.        }
VAR i         : INTEGER;
    p,r,ai,bi : REAL;
BEGIN
   p:= 0.0;
   FOR i:= LB(A) TO UB(A) DO      { A,B must have same index range }
   BEGIN
      ai:= A[i];
      bi:= B[i];
      IF ( ai*bi <= 0.0 )  OR  ( zerotest*ABS(ai) < ABS(bi) )
                     THEN   r:= 0.0
                     ELSE   r:= ABS( (ai-bi)/bi );
      IF r>p THEN p:= r;
   END;
   REL:= p;
END; { REL }


FUNCTION TOO_BAD ( VAR A : IVECTOR ) : BOOLEAN;
{ TOO_BAD := accuracy of A is far too bad }
VAR i   : INTEGER;
    bad : BOOLEAN;
BEGIN
   bad:= FALSE;
   FOR i:= LB(A) TO UB(A) DO
      bad:= bad OR (A[i].INF<-abort) AND (A[i].SUP>abort);
   TOO_BAD := bad;
END; { TOO_BAD }


PROCEDURE GUESS_ZEROES( VAR x1,x0 : RVECTOR );
{ x1 is the new, x0 the old value of an iteration. If a component of x1 }
{ has decreased by more than a factor of zerotest, then this component  }
{ is set to 0. The same is done if the sign of a component has changed. }
VAR i    : INTEGER;
    MAXX : REAL;
BEGIN                            { x0,x1 must have same index range }
  MAXX:= 0.0;
  FOR i:= LB(x1) TO UB(x1) DO IF ABS(x1[i])>MAXX THEN MAXX:= ABS(x1[i]);
  FOR i:= LB(x1) TO UB(x1) DO
    IF    ( (x0[i]*x1[i]<0.0) OR (zerotest*ABS(x1[i])<ABS(x0[i]))  )
                                   AND ( MAXX>zerotest*ABS(x1[i]) )
    THEN  x1[i]:= 0.0;
END; { GUESS_ZEROES }


PROCEDURE LSS( VAR A : RMATRIX; VAR b : RVECTOR;  VAR Y : IVECTOR;
               VAR errcode : INTEGER;
               VAR R1,R2 : RMATRIX; VAR C : IMATRIX; VAR FLAGS : INTEGER );
{ The result Y is an enclosure of the solution of   Ax=b              }
{                                                                     }
{ errcode = 0 : Y is enlosure of the solution                         }
{ errcode = 1 : no enclosure obtained, bad condition (?)              }
{ errcode = 2 : no enclosure obtained, matrix A singular (?)          }
{                                                                     }
{ FLAGS   = 0 : R1,R2,C have not yet been computed                    }
```

```
{ FLAGS   = 1 : only R1 has been computed                                }
{ FLAGS   = 2 : R1 and corresponding C have been computed                }
{ FLAGS   = 3 : R1 and R2 are computed but not the corresponding C       }
{ FLAGS   = 4 : R1, R2 and the corresponding C have been computed        }

VAR
     D              : RMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
     x0,x1,y0       : RVECTOR[LB(A)..UB(A)];
     Y1,YA,Z        : IVECTOR[LB(A)..UB(A)];
     i,j,k,err      : INTEGER;
     ready          : BOOLEAN;
     p,bound,eps    : REAL;

PROCEDURE USE_SINGLE_R;
BEGIN
{ compute approximate inverse of A }
   err:= 0;
   IF FLAGS<1 THEN
   BEGIN
      R1 := A;
      minv( R1, err );
      IF err=0 THEN FLAGS:= 1;
   END;
   IF err=0 THEN
   BEGIN

{ floating point defect iteration: result is x1 }
      bound := 100.0*sqrt_01;
      x1:= R1*b;
      k:= 0;
      REPEAT
         { iterate  x = x + R*(b-Ax)  }
         k := k + 1;
         x0:= x1;
         x1:= #*( b - A*x0 );
         x1:= #*( x0 + R1*x1 );
         p:= REL( x1, x0 );
         IF k>5 THEN bound := bound * sqrt_01;
      UNTIL ( (p>=bound) AND (k>5) )  OR  (p<delta);

      GUESS_ZEROES( x1,x0 );

{ compute enclosure y0+Y1 of the residuum b-A*x1 of the approximation x1
  and initialize  Y1:= Z:= R1*(b-A*x1), C:= I-R1*A  }

      y0:= #*( b - A*x1 );
      Y1:= ##( b - A*x1 - y0 );
      Y1:= ##( R1*y0 + R1*Y1 );
      Z := Y1;

      IF Z=NULL(Z) THEN
      BEGIN
         Y       := x1; { exact solution! (however, not necessarily unique!) }
         errcode:= 0;
         ready   := TRUE;
      END ELSE
      BEGIN
         IF FLAGS<2 THEN
         BEGIN
            C     := ##( ID(A) - R1*A );
            FLAGS:= 2;
         END;

{ interval iteration until inclusion is obtained (or max. iteration count) }
```

```
         k  := 0;
         eps:= 0.1;
         REPEAT
            IF k>=5 THEN eps := 5*eps;
            k := k + 1;
            YA:= BLOW( Y1, eps );
            Y1:= Z + C*YA;
            ready := Y1 IN YA;
         UNTIL ready OR (k>=10) OR TOO_BAD(Y1);

{ output of the result }
         IF ready THEN
         BEGIN
            Y      := x1 + Y1;
            errcode:= 0;
         END;
      END;
   END ELSE  ready:= FALSE;
END { USE_SINGLE_R };


PROCEDURE USE_DOUBLE_R;
BEGIN
{ compute approximate inverse of A }
   err:= 0;
   IF FLAGS<3 THEN
   BEGIN
      R2:= R1*A;
      MINV( R2, err );
      IF err=0 THEN
      BEGIN
         FLAGS:= 3;
         D := R2*R1;
         R2:= #*( R2*R1 - D );
         R1:= D;
      END;
   END;
   IF err=0 THEN
   BEGIN

{ floating point defect iteration: result is x1+x0 }
      bound := 100.0*sqrt_01;
      x1:= #*( R1*b + R2*b );
      x0:= #*( R1*b + R2*b - x1 );
      k:= 0;
      REPEAT
         { Iteration  x = x + (R1+R2)*(b-Ax), x = x1 + x0 }
         k := k + 1;
         y0:= #*( b - A*x1 - A*x0 );
         y0:= #*( x0 + R1*y0 + R2*y0 );
         p := REL( x1+y0, x1+x0 );
         y0:= x1 + y0;
         x0:= #*( x1 + x0 - y0 );
         x1:= y0;
         IF k>5 THEN bound := bound * sqrt_01;
      UNTIL ( (p>=bound) AND (k>5) ) OR (p<delta);

{ compute enclosure y0+Y1 of the residuum b-A*x1 of the approximation x1
  and initialize  Y1:= Z:= (R1+R2)*(b-A*x1), C:= I-(R1+R2)*A  }

      y0:= #*( b - A*x1 );
      Y1:= ##( b - A*x1 - y0 );
      Y1:= ##( R1*y0 + R2*y0 + R1*Y1 + R2*Y1 );
      Z := Y1;
```

```
      IF Z=NULL(Z) THEN
      BEGIN
         Y      := x1; { exact solution! (however, not necessarily unique!) }
         errcode:= 0;
         ready  := TRUE;
      END ELSE
      BEGIN
         IF FLAGS<4 THEN
         BEGIN
            C    := ##( ID(A) - R1*A - R2*A );
            FLAGS:= 4;
         END;

{ interval iteration until inclusion is obtained (or max. iteration count) }
         k  := 0;
         eps:= 0.1;
         REPEAT
            IF k>=5 THEN eps := 5*eps;
            k := k + 1;
            YA:= BLOW( Y1, eps );
            Y1:= Z + C*YA;
            ready := Y1 IN YA;
         UNTIL ready OR (k>=10) OR TOO_BAD(Y1);

{ output of the result }
         IF ready THEN
         BEGIN
            Y      := x1 + Y1;
            errcode:= 0;
         END ELSE
            errcode:= 1;
      END;
   END ELSE errcode:= 2;
END  { USE_DOUBLE_R };

BEGIN
   ready:= FALSE;
   IF FLAGS<3 THEN USE_SINGLE_R;
{  If no success: try again with approximate inverse R = R1+R2 of dou-
ble length  }
   IF NOT ready THEN  USE_DOUBLE_R;
END; {*** LSS ***}


PROCEDURE SQUARE_LSS( VAR A : RMATRIX; VAR b   : RVECTOR;
                      VAR Y : IVECTOR; VAR errcode : INTEGER );
{ Linear system: square matrix                         }
{ The result Y is an enclosure of the solution of Ax = b }

VAR
    R1,R2 : RMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    C     : IMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    FLAGS : INTEGER;
BEGIN
   IF dim=1 THEN  { treat trivial case separately }
   IF A[LB(A,1),LB(A,2)]=0.0 THEN errcode:= 2 ELSE
   BEGIN
      Y[LB(Y)]:= INTVAL( B[LB(B)] ) / A[LB(A,1),LB(A,2)];
      errcode:= 0;
   END ELSE
   BEGIN
      FLAGS:= 0;
      LSS( A,b,Y,errcode, R1,R2,C,FLAGS );
```

```
   END;
END; {*** SQUARE_LSS ***}


PROCEDURE OVER_LSS( VAR A : RMATRIX; VAR b    : RVECTOR;
                    VAR Y : IVECTOR; VAR errcode : INTEGER );
{ Linear system: over-determined case                              }
{ The result Y is an enclosure of the solution x von AH*A*x = AH*b, }
{ i.e. x is least squares solution of Ax=b.                        }
{                                         | A -I  ||x| |b|          }
{ From Ax=b we generate the (n+m)x(n+m)-system |     H || |=| |     }
{                                         | 0  A  ||y| |0|          }
{                                                                  }
{ Here, AH is the hermitian of A (transpose in the real case).     }
{                                                                  }
VAR BIG_A : RMATRIX[1..dim,1..dim];
    BIG_B : RVECTOR[1..dim];
    BIG_Y : IVECTOR[1..dim];
BEGIN
   BIG_A[  1..m   ,   1..n   ]:= A;
   BIG_A[  1..m   ,n+1..n+m]:= -id(m);
   BIG_A[m+1..m+n,   1..n   ]:= 0.0;
   BIG_A[m+1..m+n,n+1..n+m]:= TRANSP(A);
   BIG_B[  1..m   ]:= b;
   BIG_B[m+1..m+n]:= 0.0;

   SQUARE_LSS(BIG_A,BIG_B,BIG_Y,errcode);

   Y:= BIG_Y[1..n];
END; {*** OVER_LSS ***}


PROCEDURE UNDER_LSS( VAR A : RMATRIX; VAR b    : RVECTOR;
                     VAR Y : IVECTOR; VAR errcode : INTEGER );
{ Linear system: under-determined case                            }
{ The result Y is an enclosure of y = AH*x with A*AH*x = b        }
{ i.e. y is solution of Ay=b with minimal Euklidian norm.         }
{                                         | AH -I ||x| |0|        }
{ From Ax=b we generate the (n+m)x(n+m)-system |      || |=| |    }
{                                         | 0  A  ||y| |b|        }
{                                                                 }
{ Here, AH is the hermitian of A (transpose in the real case).    }
{                                                                 }
VAR BIG_A : RMATRIX[1..dim,1..dim];
    BIG_B : RVECTOR[1..dim];
    BIG_Y : IVECTOR[1..dim];
BEGIN
   BIG_A[  1..n   ,   1..m   ]:= TRANSP(A);
   BIG_A[  1..n   ,m+1..m+n]:= -id(n);
   BIG_A[n+1..n+m,   1..m   ]:= 0.0;
   BIG_A[n+1..n+m,m+1..m+n]:= A;
   BIG_B[  1..n   ]:= 0.0;
   BIG_B[n+1..n+m]:= b;

   SQUARE_LSS(BIG_A,BIG_B,BIG_Y,errcode);

   Y:= BIG_Y[m+1..m+n];
END; {*** UNDER_LSS ***}


GLOBAL PROCEDURE LSS( VAR A : RMATRIX; VAR b    : RVECTOR;
                      VAR Y : IVECTOR; VAR errcode : INTEGER );
{ General entry for linear system solver: decides which case to treat }
BEGIN
```

```
   errcode:= 0;
   m:= UB(A,1) - LB(A,1) + 1;
   n:= UB(A,2) - LB(A,2) + 1;

   dim:= n+m;

   IF m <> UB(b)-LB(b)+1 THEN errcode:= 3; { b : wrong dimension }
   IF n <> UB(Y)-LB(Y)+1 THEN errcode:= 4; { Y : wrong dimension }

   IF errcode=0 THEN
   IF m > n THEN  OVER_LSS(A,b,Y,errcode) ELSE   { over-determined  system }
   IF m < n THEN UNDER_LSS(A,b,Y,errcode)        { under-determined system }
            ELSE BEGIN
                   dim:= n;
                   SQUARE_LSS(A,b,Y,errcode);    { square system   }
                 END;
END {*** LSS ***};


PROCEDURE SQUARE_INV( VAR A : RMATRIX; VAR Y : IMATRIX;
                      VAR errcode : INTEGER );
{ Inverse matrix: square case                            }
{ The result Y is an enclosure of the solution of AY = I }
VAR
    R1,R2       : RMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    C           : IMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    b           : RVECTOR[LB(A)..UB(A)];
    FLAGS,i,err : INTEGER;
BEGIN
   IF dim=1 THEN  { treat trivial case separately }
   IF A[LB(A,1),LB(A,2)]=0.0 THEN errcode:= 2 ELSE
   BEGIN
      Y[LB(Y,1),LB(Y,2)]:= intval(1.0) / A[LB(A,1),LB(A,2)];
      errcode:= 0;
   END ELSE
   BEGIN
      FLAGS:= 0;
      err  := 0;
      b    := 0.0;
      FOR i:= LB(A,1) TO UB(A,1) DO
      BEGIN
         b[i]:= 1.0;
         LSS( A,b,Y[LB(Y,1)+i-LB(A,1)],errcode, R1,R2,C,FLAGS );
         if errcode>err then err:= errcode;
         b[i]:= 0.0;
      END;
      errcode:= err;
      Y:= TRANSP(Y);
   END;
END; {*** SQUARE_INV ***}


PROCEDURE OVER_INV( VAR A : RMATRIX; VAR Y : IMATRIX;
                    VAR errcode : INTEGER );
{ Inverse matrix: over-determined case                          }
{ The result Y is an enclosure of the pseudo inverse A+ of A.  }
{                                                               }
{                   +     H  -1 H                               }
{ If m>n then Y = A  = (A A)  A   is solution of:               }
{                                                               }
{  | A  -I ||Y|   |I|     ( A : mxn          )                  }
{  |     H || | = | | ,   ( X = AY-I : mxm   )                  }
{  | 0  A  ||X|   |0|     ( Y = A+   : nxm   )                  }
{                         ( right hand side: )                  }
```

```
{                         ( I : mxm, 0 : nxm )                 }
VAR BIG_A,R1,R2 : RMATRIX[1..dim,1..dim];
    C           : IMATRIX[1..dim,1..dim];
    BIG_B       : RVECTOR[1..dim];
    BIG_Y       : IVECTOR[1..dim];
    FLAGS,j,err : INTEGER;
BEGIN
   BIG_A[  1..m   ,  1..n   ]:= A;
   BIG_A[  1..m   ,n+1..n+m]:= -id(m);
   BIG_A[m+1..m+n,  1..n   ]:= 0.0;
   BIG_A[m+1..m+n,n+1..n+m]:= TRANSP(A);
   BIG_B:= 0.0;

   FLAGS:= 0;
   err  := 0;
   FOR j:= 1 TO m DO
   BEGIN
      BIG_B[j]:= 1.0;
      LSS( BIG_A,BIG_B,BIG_Y,errcode, R1,R2,C,FLAGS );
      if errcode>err then err:= errcode;
      BIG_B[j]:= 0.0;
      Y[*,LB(Y,2)+j-1]:= BIG_Y[1..n];
   END;
   errcode:= err;
END; {*** OVER_INV ***}


PROCEDURE UNDER_INV( VAR A : RMATRIX; VAR Y : IMATRIX;
                     VAR errcode : INTEGER );
{ Inverse matrix: under-determined case                        }
{ The result Y is an enclosure of the pseudo inverse A+ of A.  }
{                                                              }
{                  +    H   H -1                               }
{ If m<n then Y = A  = A (AA )   is solution of:               }
{                                                              }
{     H       H                                                }
{  | A  -I ||Y |    |I|    ( A : mxn          )                }
{  |       || H| =  | | ,  ( X = YA-I : nxn   )                }
{  | 0   A ||X |    |0|    ( Y = A+   : nxm   )                }
{                          ( right hand side: )                }
{                          ( I : nxn, 0 : mxn )                }
{                                                              }
VAR BIG_A,R1,R2 : RMATRIX[1..dim,1..dim];
    C           : IMATRIX[1..dim,1..dim];
    BIG_B       : RVECTOR[1..dim];
    BIG_Y       : IVECTOR[1..dim];
    FLAGS,i,err : INTEGER;
BEGIN
   BIG_A[  1..n   ,  1..m   ]:= TRANSP(A);
   BIG_A[  1..n   ,m+1..m+n]:= -id(n);
   BIG_A[n+1..n+m,  1..m   ]:= 0.0;
   BIG_A[n+1..n+m,m+1..m+n]:= A;
   BIG_B:= 0.0;

   FLAGS:= 0;
   err  := 0;
   FOR i:= 1 TO n DO
   BEGIN
      BIG_B[i]:= 1.0;
      LSS( BIG_A,BIG_B,BIG_Y,errcode, R1,R2,C,FLAGS );
      if errcode>err then err:= errcode;
      BIG_B[i]:= 0.0;
      Y[LB(Y,1)+i-1]:= BIG_Y[1..m];
   END;
```

```
   errcode:= err;
END; {*** UNDER_INV ***}


GLOBAL PROCEDURE INV( VAR A : RMATRIX; VAR Y : IMATRIX;
                      VAR errcode : INTEGER );
{ General entry for matrix inversion: decides which case to treat }
BEGIN
   errcode:= 0;
   m:= UB(A,1) - LB(A,1) + 1;
   n:= UB(A,2) - LB(A,2) + 1;

   dim:= m+n;

   IF n<>UB(Y,1)-LB(Y,1)+1 THEN errcode:= 3; { Y : wrong number of rows    }
   IF m<>UB(Y,2)-LB(Y,2)+1 THEN errcode:= 4; { Y : wrong number of columns }

   IF errcode=0 THEN
   IF m > n THEN  OVER_INV(A,Y,errcode) ELSE    { over-determined system  }
   IF m < n THEN UNDER_INV(A,Y,errcode)         { under-determined system }
            ELSE BEGIN
                   dim:= n;
                   SQUARE_INV(A,Y,errcode);     { square System   }
                 END;
END; {***  INV  ***}

BEGIN
   { module body empty }
END.
```

Concluding this section we want to indicate which changes have to be done in the preceding PASCAL-XSC program code in order to get programs that treat the cases of *interval*, *complex* and *complex interval* input data.

First we consider the case of interval data, i.e. the input matrix A is an interval matrix and the right hand side b is an interval vector.

We construct a new module `ilss` from the above module `lss` by applying the changes which we describe now.

This are the changes in the first (local) procedure `LSS`:

- In the procedure head make `A` an `IMATRIX` and `b` an `IVECTOR`.
- In the declaration part add variables `AM` of type `RMATRIX` and `bm` of type `RVECTOR` which will be used to hold the midpoints of `A` and `b`.
- At the beginning of the procedure body add the statements `AM:= MID(A);  bm:= MID(b);` .
- In the local procedure `USE_SINGLE_R` replace in each of the three statements `R1:= A;` `x1:= R1*b;` and `x1:= #*( b - A*x0 );` the variables `A` by `AM` and `b` by `bm`.
- Also in `USE_SINGLE_R` replace `y0:= #*( b - A*x1 );` by `y0:= MID( ##( b - A*x1 ) );`

- In the local procedure `USE_DOUBLE_R` replace in each of the four statements `R2:= R1*A;` `x1:= #*( R1*b + R2*b );` `x0:= #*( R1*b + R2*b - x1 );` and `y0:= #*( b - A*x1 - A*x0 );` the variables `A` by `AM` and `b` by `bm`.
- Also in `USE_DOUBLE_R` replace `y0:= #*( b - A*x1 );` by `y0:= MID( ##( b - A*x1 ) );`

The other procedures in the module must be modified as follows:

- In the heads of the procedures `SQUARE_LSS`, `OVER_LSS`, `UNDER_LSS`, `LSS`, `SQUARE_INV`, `OVER_INV`, `UNDER_INV` and `INV` make `A` an `IMATRIX` and `b` an `IVECTOR`.
- In `SQUARE_LSS` and in `SQUARE_INV` replace the test `A[LB(A,1),LB(A,2)]=0.0` by the test `0.0 IN A[LB(A,1),LB(A,2)]` and delete the conversion function `INTVAL` in the special case `dim=1` in both procedures.
- In the procedures `OVER_LSS`, `UNDER_LSS`, `OVER_INV` and `UNDER_INV` change the types of the local variables `BIG_A` and `BIG_B` to the types `IMATRIX` and `IVECTOR`.
- In procedure `SQUARE_INV` change the type of the local variable `b` from `RVECTOR` to `IVECTOR`.

Next, we turn to the case of a complex input matrix A and a complex input vector b. In this case we also have to modify the procedure `MINV` such that an approximate inverse of a complex matrix can be computed. We write a new module `clssaprx` by copying `lss_aprx` with the following modifications:

- Instead of using module `mv_ari` we use modules `c_ari,mvc_ari` for complex and complex matrix/vector arithmetic.
- The types of the parameter `A` of `MINV` and the local variables `r` and `h` must be replaced by their equivalent complex types `CMATRIX`, `CVECTOR` and `COMPLEX`.
- The two occurrences of the conversion function `RVECTOR` in the body of `MINV` must be replaced by the conversion function `CVECTOR`.

A module `clss` will now be constructed as a copy of module `lss` by applying the following changes:

- Replace the `USE` line by `USE i_ari,c_ari,ci_ari,mv_ari,mvc_ari, mvci_ari,clssaprx;`
- Replace *all* occurrences of `RMATRIX` by `CMATRIX`, `RVECTOR` by `CVECTOR`, `IMATRIX` by `CIMATRIX` and `IVECTOR` by `CIVECTOR` throughout the module.
- In function `REL` change the type of `ai,bi` from `REAL` to `COMPLEX` and delete the test `ai*bi <= 0.0` without replacement.

- In function `TOO_BAD` replace statement
```
bad:= bad OR (A[i].INF<-abort) AND (A[i].SUP>abort);
```
by
```
bad:= bad OR (A[i].RE.INF<-abort) AND (A[i].RE.SUP>abort)
          OR (A[i].IM.INF<-abort) AND (A[i].IM.SUP>abort);
```

- In function `GUESS_ZEROES` delete the test `x0[i]*x1[i]<0.0` without replacement.

- In the procedures `OVER_LSS`, `UNDER_LSS`, `OVER_INV` and `UNDER_INV` we must replace all function calls to the function `TRANSP` by calls to the function `HERM`. Additionally, in `UNDER_INV` the statement `Y[LB(Y,1)+i-1]:= BIG_Y[1..m];` must be replaced by `Y[LB(Y,1)+i-1]:= CONJ( BIG_Y[1..m] );`.

Finally a new module `cilss` can be constructed from the module `ilss` by applying exactly the same changes as for the conversion from `lss` to `clss`.

2.1.6 Test Results

A very well known set of ill conditioned test matrices for linear system solvers are the $n \times n$ Hilbert matrices $H_n := (\frac{1}{i+j-1})$. As a test problem we report the results of our program for the linear systems $H_n x = e_1$ where e_1 is the first canonical unit vector. Thus the solution x is the first column of the inverse H_n^{-1} of the Hilbert matrix H_n. We give results for the cases $n = 10$ and $n = 20$. Since the elements of these matrices are rational numbers which can not be stored exactly in floating point, we do not solve the given problems directly but rather we multiply the system by the least common multiple lcm_n of all denominators in H_n. Then the matrices will have integer entries which makes the problem exactly storable in IEEE floating point arithmetic. For $n = 10$ we have $lcm_{10} = 232792560$ and for $n = 20$ we have $lcm_{20} = 5342931457063200$.

For the system $(lcm_{10}H_{10})x = (lcm_{10}e_1)$ the program computes the result:

x_1	1.000000000000000E+002
x_2	-4.950000000000000E+003
x_3	7.920000000000000E+004
x_4	-6.006000000000000E+005
x_5	2.522520000000000E+006
x_6	-6.306300000000000E+006
x_7	9.609600000000000E+006
x_8	-8.751600000000000E+006
x_9	4.375800000000000E+006
x_{10}	-9.237800000000000E+005

which is the exact solution of this ill conditioned system.

For the system $(lcm_{20}H_{20})x = (lcm_{20}e_1)$ the program computes the enclosures (here an obvious short notation for intervals is used):

x_1	${}^{4.000000000000001}_{3.999999999999999}E+002$
x_2	$-7.9^{7999999999999998}_{8000000000000002}E+004$
x_3	$5.266^{800000000001}_{799999999999}E+006$
x_4	$-1.71609^{8999999999}_{9000000001}E+008$
x_5	$3.2949100^{80000001}_{79999999}E+009$
x_6	$-4.118637^{599999999}_{600000001}E+010$
x_7	$3.5694859^{20000001}_{19999999}E+011$
x_8	$-2.23730278^{1999999}_{2000001}E+012$
x_9	$1.044074631^{600001}_{599999}E+013$
x_{10}	$-3.7006645275^{59999}_{60001}E+013$
x_{11}	$1.0092721438^{80001}_{79999}E+014$
x_{12}	$-2.1332343041^{09999}_{10001}E+014$
x_{13}	$3.5006921913^{60001}_{59999}E+014$
x_{14}	$-4.4431862428^{79999}_{80001}E+014$
x_{15}	$4.31623806451^{2001}_{1999}E+014$
x_{16}	$-3.1472569220^{39999}_{40001}E+014$
x_{17}	$1.6661948410^{80001}_{79999}E+014$
x_{18}	$-6.044040109^{799999}_{800001}E+013$
x_{19}	$1.343120024^{400001}_{399999}E+013$
x_{20}	$-1.378465288^{199999}_{200001}E+012$

which is an extremely accurate enclosure for the exact solution. (The exact solution components are the integers within the computed intervals).

2.1.7 Notes and References

In the program the case $n = 2$ should of course be computed separately, as it is done for $n = 1$. This concerns the procedures `SQUARE_LSS` and `SQUARE_INV` in module lss as well as `MINV` in lss_aprx. We have omitted these cases to keep the procedures somewhat shorter. Even $n = 3$ might be worth to be programmed directly.

The form of the result of iterated defect correction in the algorithm

$$x \in \tilde{x} + [y]$$

is a special case of a so-called staggered correction format which has the more general form

$$x \in \sum_{i=1}^{k} x_i + [y]$$

where the sum of the x_i is an approximation of x and $[y]$ an enclosure of its error. Such a representation can easily be obtained by a modification of our algorithm in such a way that several iterated defect correction steps are performed. The staggered correction format will be extensively discussed in Section 6.

If the algorithm is modified in the way that it produces output in staggered correction format then a further modification is almost trivial such that also input is possible in staggered format i.e. the coefficient matrix A and the right hand side b may be in staggered form. Then we have a verifying multi-precision linear system solver.

Such a linear system solver in staggered format can be applied to solve the least squares problem and compute the pseudo inverse in a different and cheaper way as we have done it in our program. For the least squares problem we proceed as follows (see [308]): In the case $m \geq n$ we compute $B := A^H A$ and $d := A^H b$ in double length (i.e. B and d have staggered format) and solve $Bx = d$ with the staggered linear system solver. In the case $m \leq n$ we compute $B := AA^H$ in double length and solve $Bx = b$ with the staggered linear system solver. Then, finally, $y = A^H x$ is the desired solution (which must be computed in double length).

The methods presented in this section were originally developed by S. Rump, [280], [285], some ideas go back to Wilkinson, [333].

In the meantime some newer methods exist which use LU-Factorization with backward error estimation for the solution of full triangular systems, [153]. These algorithms can be adapted to the condition of the system and use less computing time as ours in the case of well conditioned problems.

Also for symmetric matrices there exist special methods which give better results especially in the case of an interval matrix A and which have somewhat less overhead.

For symmetric and positive definite matrices a variant of the Cholesky method has been investigated by [19].

Similarly for M-matrices and H-matrices different methods exist, e.g. the interval Gauss elimination can be applied in these cases. For details see [11], [13], [242].

Another wide area of methods are iterative methods which are not directly based on defect correction. See also [238], [239], [241].

Finally we note that our algorithm is not well suited for large systems which are sparse or which have banded structure since the approximate inverse R will be in general a full matrix. Special methods for these cases are discussed in the next section.

2.2 Systems with Band Matrices

2.2.1 Introduction

For linear systems where the coeffient matrix A has band structure the algorithm from Section 2.1 can be used without modification of course. However, since an approximate inverse R is used there this would result in a large overhead of storage and computation time, especially if the bandwidth of A is small compared with its dimension. To reduce this overhead, we will replace the approximate inverse by an approximate LU-decomposition of A which needs memory of the same order of magnitude only as A itself. Then we will have to solve systems with triangular banded matrices (containing point data) in interval arithmetic. This seems to be a trivial task and several methods have been developed using such systems and simple forward and backward substitution in interval arithmetic, see e.g. [177], [178], [179], [78], [79], [295]. However, at this point there appears suddenly a very unpleasant effect which makes the computed intervals blow up very rapidly in many cases. This effect is known in literature as *wrapping effect* (see e.g. [247], [257]) and was recognized first in connection with the verified solution of ordinary initial value problems. However it is a common problem within interval arithmetic and may show up whenever computations in $I\!R^n$, $n > 1$, are performed, e.g. if an interval vector is multiplied repeatedly by matrices or more general if a function $f : I\!R^n \to I\!R^n$ is applied repeatedly to an argument x, $x_n = f(f(\cdots f(f(x))\cdots))$. It should be noted that the wrapping effect is not related to roundoff errors or any other sources of errors in an algorithm (like truncation errors, discretization errors and the like) but it is introduced solely by interval arithmetic itself (though any additional errors may contribute to an increase of the wrapping effect).

Some simple examples illustrate this effect very drastically:

Example 2.1:
This example is the classical example for which the wrapping effect was discovered first. It comes from the solution of the initial value problem $u'' + u = 0, u(0) = u_0, u'(0) = u_1$, see [247], [207]. With the matrix

$$A_\phi := \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix}$$

consider the following simple recursion in $I\!R^2$:

$$x_{n+1} = A_\phi x_n, \quad n \geq 0, \quad x_0 \in [x_0] := \epsilon \begin{pmatrix} [-1,1] \\ [-1,1] \end{pmatrix} \quad \epsilon > 0. \tag{2.17}$$

For each point vector $x_0 \in [x_0]$ we have obviously the solution

$$x_n = A_\phi^n x_0 = \begin{pmatrix} \cos n\phi & \sin n\phi \\ -\sin n\phi & \cos n\phi \end{pmatrix} x_0, \; n \geq 0 \tag{2.18}$$

i.e. the vector x_n is constantly rotating around the origin by an angle ϕ each iteration. Thus the solution set $\{x_n | x_0 \in [x_0]\}$ is just the original square $[x_0]$ rotating constantly around the origin. Trivially this set remains bounded for all $n \geq 0$.

In contrast if we carry out the iteration (2.17) in interval arithmetic (with exact computations, no roundoff errors)

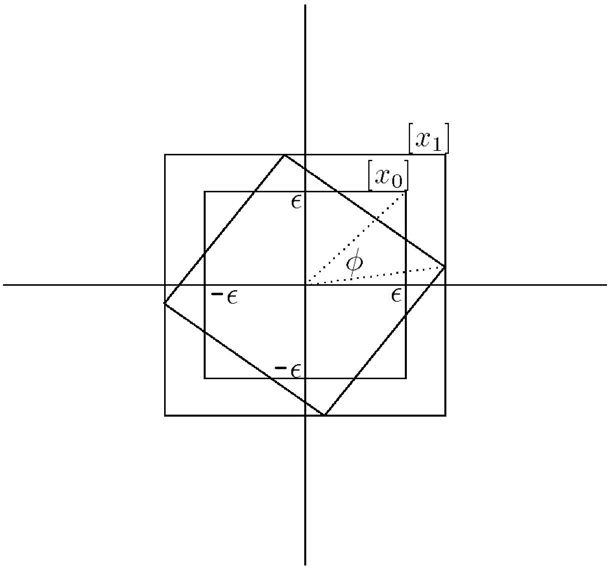
$$[x_{n+1}] = A_\phi [x_n] \tag{2.19}$$

where all operations are interval operations now, then we obtain

$$[x_1] \;=\; \begin{pmatrix} [-\epsilon,\epsilon]\cos\phi + [-\epsilon,\epsilon]\sin\phi \\ [-\epsilon,\epsilon]\sin\phi + [-\epsilon,\epsilon]\cos\phi \end{pmatrix} = \epsilon(|\sin\phi| + |\cos\phi|) \begin{pmatrix} [-1,1] \\ [-1,1] \end{pmatrix}$$

$$
\begin{aligned}
[x_2] &= \epsilon(|\sin\phi| + |\cos\phi|)^2 \begin{pmatrix} [-1,1] \\ [-1,1] \end{pmatrix} \\
&\ \vdots \\
[x_n] &= \epsilon(|\sin\phi| + |\cos\phi|)^n \begin{pmatrix} [-1,1] \\ [-1,1] \end{pmatrix} \quad .
\end{aligned}
$$

The interval vectors $[x_n]$ are of course enclosures for the bounded true solution sets $\{x_n | x_0 \in [x_0]\}$. However, since $|\sin\phi| + |\cos\phi| > 1$ for all $\phi \neq k\pi/2$ we see that the diameter of this enclosure diverges to ∞. This is the aforementioned wrapping effect which has a simple geometric explanation: the interval vector $[x_0]$ is rotated by an angle of ϕ around the origin and then enclosed ('wrapped') by the smallest possible interval vector $[x_1]$ whose diameter is $|\sin\phi| + |\cos\phi|$ times as large as that of $[x_0]$. The same process is repeated in each of the following iteration steps. We note that this wrapping occurs even though we use an optimal enclosure in each step. The following picture illustrates this geometric process.



This iteration can easily be rewritten as a system of linear equations

$$
Ax = b \tag{2.20}
$$

with a matrix A which is lower triangular and banded. We write $A \in I\!R^{(2n+2)\times(2n+2)}$ and $x, b \in I\!R^{2n+2}$ in the following block notation with 2×2-blocks in A, I being the 2×2-identity matrix, and 2-vectors in x and b:

$$
\begin{aligned}
A &= \begin{pmatrix} I & & & \\ A & -I & & \\ & \ddots & \ddots & \\ & & A & -I \end{pmatrix} \\
x &= (x_0, x_1, \ldots, x_n)^T \\
b &= (x_0, 0, \ldots, 0)^T
\end{aligned}
$$

Obviously solving this block system by forward substitution in interval arithmetic is equivalent to the computation of the iteration in (2.19). Thus we have a first example that simple forward substitution in an interval setting can produce unacceptably large overestimations.

Example 2.2:
A similar example as the previous one, now with a full lower triangular matrix is as follows. Let A be the $n \times n$-matrix

$$A = \begin{pmatrix} 1 & & & & & \\ 2 & 1 & & & & \\ 3 & 2 & 1 & & & \\ 4 & 3 & 2 & 1 & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ n & \cdots & 4 & 3 & 2 & 1 \end{pmatrix} .$$

The inverse A^{-1} can easily be verified to be the banded lower triangular matrix

$$A = \begin{pmatrix} 1 & & & & \\ -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \end{pmatrix} .$$

If we choose as right hand side the interval vector

$$[b] = \epsilon([0,1],[0,1],\ldots,[0,1])^T$$

then the optimal enclosure of the exact solution set $\{x = A^{-1}b | b \in [b]\}$ is

$$[x]_{opt} = \epsilon([0,1],[-2,1],[-2,2],\ldots,[-2,2])^T$$

which is identical to $[-2\epsilon, 2\epsilon]$ in all components but the first and second.

However, solving $Ax = [b]$ by forward substitution in interval arithmetic yields the following enclosures

$$x \in \epsilon \begin{pmatrix} [0,1] \\ [-2,1] \\ [-5,5] \\ [-17,17] \\ [-58,58] \\ [-198,198] \\ [-676,676] \\ [-2308,2308] \\ [-7880,7880] \\ [-26904,26904] \\ \vdots \end{pmatrix}$$

whose diameters are obviously diverging. This is an example where a blow up of the enclosing intervals occurs without having a nice and obvious geometric interpretation.

We note that also the system $By = [b]$ with the inverse matrix $B := A^{-1}$ as coefficient matrix shows a similar behaviour: the solution set $\{y = B^{-1}b = Ab | b \in [b]\}$ is unbounded here, but nevertheless the enclosures obtained by interval forward substitution still overestimate this set by an even far greater amount as can be easily checked. This system has also a great similarity with the following example.

Example 2.3:
The last example starts with a scalar, linear difference equation. The well-known Chebychev polynomials $T_n(x)$ can be defined by the second order recurrence relation

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{n+1}(x) - 2xT_n(x) + T_{n-1}(x) = 0 \ .$$

If we wish to compute an enclosure of $T_n(x)$ for some value x_0 and a high value of n then we can use this difference equation and compute the desired value in interval arithetic. However, this computation is nothing else but interval forward substitution for the system

$$\begin{pmatrix} 1 & & & & \\ 0 & 1 & & & \\ 1 & -2x_0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2x_0 & 1 \end{pmatrix} \begin{pmatrix} T_0(x_0) \\ T_1(x_0) \\ T_2(x_0) \\ \vdots \\ T_n(x_0) \end{pmatrix} = \begin{pmatrix} 1 \\ x_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} .$$

This is a system with point data only, however in interval floating-point arithmetic we will get sooner or later roundoff errors which introduce intervals with nonzero diameters. Then as in the last examples here also a very fast blow up of the enclosures will occur. If the argument x_0 is not a floating-point number, then such intervals will enter the computation right from the beginning. If we take e.g. $x_0 = 0.99$ then we obtain the following output from a short PASCAL-XSC program (IEEE double precision) for the values of n and the enclosures for $T_n(0.99)$:

```
 n                      T[n](0.99)

 2   [  9.601999999999999E-001,  9.602000000000007E-001 ]
 3   [   9.11195999999999E-001,   9.11196000000002E-001 ]
 4   [   8.43968079999998E-001,   8.43968080000004E-001 ]
 5   [    7.5986079839999E-001,    7.5986079840001E-001 ]
 6   [    6.6055630083198E-001,    6.6055630083202E-001 ]
 7   [     5.480406772473E-001,     5.480406772474E-001 ]
 8   [     4.245642401176E-001,     4.245642401179E-001 ]
 9   [     2.925965181856E-001,     2.925965181861E-001 ]
10   [      1.54776865889E-001,      1.54776865891E-001 ]
15   [      -5.246430527E-001,       -5.246430525E-001 ]
20   [       -9.52088247E-001,        -9.52088240E-001 ]
25   [         -9.222663E-001,          -9.222657E-001 ]
```

```
30   [          -4.496E-001,          -4.494E-001 ]
35   [            2.3E-001,             2.5E-001 ]
40   [            6.9E-001,             9.3E-001 ]
45   [           -8.0E+000,             1.0E+001 ]
50   [           -7.1E+002,             7.2E+002 ]
```

Of course the values of the Chebychev polynomials can be computed in a much easier way, since $T_n(x) = \cos(n \arccos x)$. This example should only demonstrate the difficulties that can arise with difference equations even as simple as this one. Nevertheless this example still is relevant since many other functions can be computed by use of certain difference equations (e.g. Bessel functions where the recursion $J_{n+1}(x) - \frac{2n}{x} J_n(x) + J_{n-1}(x) = 0$ is similarly sensitive to the wrapping effect).

These examples show clearly that it is important to find good methods for enclosing the solutions of triangular system of equations. Also it is obvious that difference equations and banded triangular systems are very closely related. Thus it can be expected that a method that works well with either one type should also work well with the other type of systems. This relationship between linear difference equations and linear systems with triangular matrix will be the basis for the method which we present in this Section for difference equations and for systems with banded matrices.

In Section 2.2.2 we will first treat the case of difference equations. We will demonstrate a method for first order vector difference equations (such as in Example 2.1) which reduces the wrapping effect to a large extent by enclosing the solution in suitably chosen coordinate systems. Scalar difference equations of higher order than one (as in Example 2.3) are then reformulated as linear first order matrix-vector difference equations. In this way the benefit of the reduction of the wrapping effect carries over to scalar difference equations of higher order also. Finally in Section 2.2.3 we will apply the same technique to linear systems with banded matrices (see Example 2.2) by rewriting them as linear difference equations and applying the methods of Section 2.2.2.

Other approaches to large systems – banded and sparse – can be found in [292]. An investigation of the wrapping effect within our context of linear systems is contained in [257].

2.2.2 Difference Equations

2.2.2.1 Theoretical Background

We start this section by discussing an enclosure method for first order vector difference equations of the following form

$$y_{i+1} = A_i y_i + d_i, \quad i \geq 0, \quad \text{with} \quad y_0 \text{ given.} \tag{2.21}$$

Here, the y_i and d_i are m-vectors and the A_i are $m \times m$-matrices ($i \geq 0$).

This equation has an immediate geometric interpretation in $I\!R^m$ since it is an affine mapping of y_i to y_{i+1}. From Example 2.1 we know that we may run into the wrapping effect if we compute (2.39) in interval arithmetic. However, here we know a geometric explanation of this effect and therefore can take counter-measures using this information. Assume that at some index i of the iteration we have an interval vector $[y_i]$ as the result of (2.39) then $\{y_{i+1}|y_i \in [y_i]\}$ will be the affine image of $[y_i]$, i.e. this set will be a parallel-epiped. If we try to compute the enclosure as parallel-epipeds rather than as interval vectors then we can avoid the wrapping completely. Unfortunately this is true in theory only, since the roundoff errors on a computer bring back this effect again, see e.g. [257]. Nevertheless using parallel-epipeds is of great advantage also when computing in floating-point arithmetic.

A parallel-epiped can easily be represented on a computer by use of a (regular) basis matrix B an interval vector $[z]$ and a point vector $\tilde{y}$. Then the point set

$$P := P(B, [z], \tilde{y}) := \{\tilde{y} + Bz | z \in [z]\} \tag{2.22}$$

is a parallel-epiped with one corner being $\tilde{y}$ and with edges parallel to the column vectors of B.

A somewhat simpler form is

$$P := P(B, [z]) := \{Bz | z \in [z]\} \tag{2.23}$$

where we have omitted $\tilde{y}$ which can be compensated for by translating the interval vector $[z]$ suitably. The form (2.23) is less suited if P contains points with large norm but is relatively small in diameter. In this case (2.22) is to be preferred. However for P containing small vectors only, (2.23) is adequate and simplifies computations. We will use the latter form since we are primarily concerned to enclose small error terms in parallel-epipeds by rewriting the solution y_i of (2.39) as a floating-point approximation $\tilde{y}_i$ plus an error $\hat{y}_i$ which will be enclosed in an interval $[\hat{y}_i]$:

$$y_i = \tilde{y}_i + \hat{y}_i \tag{2.24}$$

with an approximation

$$\tilde{y}_{i+1} \approx A_i \tilde{y}_i + d_i \tag{2.25}$$

such that

$$\hat{y}_{i+1} = A_i \hat{y}_i + d_i + (A_i \tilde{y}_i - \tilde{y}_{i+1}) \ . \tag{2.26}$$

For $i = 0$ we choose $\tilde{y}_0 = y_0$ if y_0 is a vector of floating-point numbers, otherwise we round y_0 to a floating-point vector and enclose the roundoff error $\hat{y}_0$ in an interval vector $[\hat{y}_0] = \Diamond(y_0 - \tilde{y}_0)$.

To compute an enclosure of the error $\hat{y}_{i+1}$ we now arrange the computation in such a way that we enclose the result of each iteration in a parallel-epiped, i.e. $\hat{y}_{i+1} \in P(B_{i+1}, [z_{i+1}])$. In each step we introduce a new basis matrix B_{i+1} which we use to represent the resulting parallel-epiped.

Introducing the abbreviation

$$\hat{d}_i = d_i + (A_i \tilde{y}_i - \tilde{y}_{i+1})$$

we replace the iteration (2.26) by

$$\begin{aligned} [z_{i+1}] &:= (B_{i+1}^{-1} A_i B_i)[z_i] + B_{i+1}^{-1}\hat{d}_i, \quad i \geq 0 \\ [\hat{y}_{i+1}] &:= B_{i+1}[z_{i+1}] \end{aligned} \tag{2.27}$$

where B_{i+1} is a regular basis matrix which may be chosen in several different ways to be dicussed later. For $i = 0$ we set $B_0 = I$ and $[z_0] = 0$ or if $\tilde{y}_0 \neq y_0$ then $[z_0] = [\hat{y}_0]$.

The representation of $\hat{y}_i$ as a parallel-epiped $P(B_i, [z_i])$ is used in the first line of (2.27) to compute the interval vector $[z_{i+1}]$ for the parallel-epiped $P(B_{i+1}, [z_{i+1}])$ of the next iteration and in the second line it is used to compute an optimal enclosure $[\hat{y}_{i+1}]$ of this parallel-epiped. This enclosure however is used only for projecting the result to orthogonal coordinates (which is convenient for output purposes) and *not* for further computations which would bring back again the wrapping effect.

We see that in this modified algorithm in each step we need an inverse of the new basis matrix B_{i+1}. Although we will always choose basis matrices which are exactly representable on the computer we still cannot get exact inverses in floating-point so that we have to compute an enclosure of B_{i+1}^{-1} instead, i.e. we use Algorithm 2.2.4 of Section 2.1 for this purpose. We also note that the product $A_i B_i$ cannot be computed exactly on the computer so that we will in general obtain an interval matrix instead even if A_i is also a floating-point matrix. In many cases even A_i will already be an interval matrix.

Next, we turn to the choice of the basis matrices B_i. There are many possibilities for this choice some of which we will discuss now. A trivial possibility is

$$B_i := I, \quad i \geq 0\ . \tag{2.28}$$

In this simple special case our new algorithm (2.27) reduces to the previous ordinary one, (2.39), where we computed just an interval vector as enclosing set in each iteration.

A more sophisticated choice is

$$B_0 := I, \quad B_{i+1} := \mathrm{mid}(\Diamond A_i B_i)\ , i \geq 0, \tag{2.29}$$

where we have taken into account that $A_i B_i$ is an interval matrix in general on the computer, as discussed above. This choice means that B_{i+1} approximates the product $A_i \cdots A_0$. However, as we can see in the very common case when the A_i are independent of i then this product is the power A^{i+1} which becomes very ill-conditioned if A has eigenvalues with different real parts, since then the eigenvector for the largest eigenvalue dominates the columns of A^{i+1} which are therefore close to being linearly dependent.

Thus we must expect the matrices B_{i+1} to become ill-conditioned or even singular very soon. Since we need the inverse of this matrix in (2.27) we must expect the iteration to yield rapidly growing enclosures or even to break down after few iterations. Only in the case where the A_i are orthogonal or nearly orthogonal the matrices B_{i+1} will stay well behaved for many iteration. E.g. in Example 2.1 this choice of B_{i+1} would be adequate and almost eliminate the wrapping effect, since there A is an orthogonal matrix independent of i.

The problems which are caused by this choice of B_{i+1} are eliminated if we force B_{i+1} to be a regular and even well-conditioned matrix by choosing it always as an orthogonal matrix. I.e. instead of the choice (2.29) we first carry out a QR-decomposition of the matrix $\mathrm{mid}(\Diamond A_i B_i)$ and take only the orthogonal part Q as new basis matrix:

$$B_{i+1} := Q_{i+1}, \quad \text{where} \quad \tilde{B}_{i+1} = \mathrm{mid}(\Diamond A_i B_i), \quad \tilde{B}_{i+1} \approx Q_{i+1} R_{i+1} \ . \tag{2.30}$$

Here $Q_{i+1}R_{i+1}$ is an approximate QR-decomposition of $\tilde{B}_{i+1}$. Q is almost orthogonal and R is an upper triangular matrix. It is completely sufficient to carry out this approxiate decomposition in floating-point arithmetic without verification. We do not present the details here since we will use a standard numerical algorithm which gives Q as the product of Housholder matrices (see e.g. [309] or [308]). This algorithm is listed in the next subsection. Now we know that B_{i+1} is almost orthogonal and thus a regular and well-conditioned matrix. Geometrically this choice of the basis matrices means that we take only rotated hyper-rectangles as enclosing sets thus avoiding long and thin parallel-epipeds whose corners depend very sensitively on changes in the direction of the edges.

The choice of orthogonal or in our case almost orthogonal basis matrices B_i has another advantage here. As we saw in (2.27) we need an enclosure of the inverse of B_i. For an orthogonal matrix Q we have $Q^{-1} = Q^T$ and thus for an almost orthogonal matrix Q the transposed matrix is a good approximation of the inverse $Q^T \approx Q^{-1}$. In this case we have even a very good rigorous estimate of the error by use of the Neumann series i.e. we do not need the algorithms from Section 2.1 for enclosing the inverse, but rather we can compute an enclosure of Q^{-1} directly.

For an almost orthogonal matrix we have

$$||I - QQ^T||_\infty = q << 1$$

where we take the $||\cdot||_\infty$ norm for convenience.

If we compute Q in floating-point arithmetic by Housholder transformations, say, then we can observe that q is only little larger than the order *eps* of the relative machine accuracy.

An enclosure $[Q^{-1}]$ of the inverse of Q now follows immediately from the Neumann series

$$Q^{-1} \in [Q^{-1}] := Q^T + [-\epsilon, \epsilon] \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & & \vdots \\ 1 & \cdots & 1 \end{pmatrix} \tag{2.31}$$

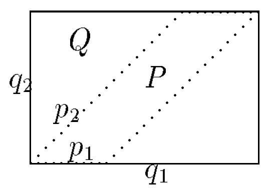
where

$$\epsilon = \frac{q}{1-q} \, ||Q^T||_\infty \ . \tag{2.32}$$

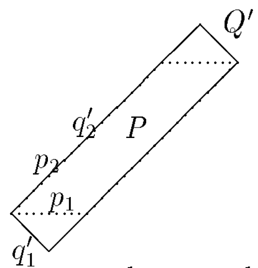
There is an additional trick, however, which we should also apply since otherwise the resulting intervals would very oddly depend on the order of the rows in A_i i.e. on the order of the components in equation (2.39) since reordering the rows of A_i changes the matrix Q_i which is computed in (2.30).

We avoid this by reordering the columns of $\tilde{B}_{i+1}$ prior to the QR-decomposition in the following way. From (2.27) we see that the shape of the parallel-epiped $P(\tilde{B}_{i+1}, [z_i])$ is an approximation of the shape of the solution set in the $(i+1)$-st step (we have ommited the translational part d_i which does not alter this shape). Therefore the edges of the solution set are approximately parallel to the columns of $\tilde{B}_{i+1}$. Then the length of each edge is the length of the corresponding column multiplied by the diameter of the same component in the interval vector $[z_i]$. Now orthogonalizing $\tilde{B}_{i+1}$ means that we start with the edge in direction of the first column, leave it fixed and make the edge in direction of the second column orthogonal to the first one. The same will be done with the other edges. From this geometric interpretation we can deduce that we will obtain a smaller enclosure if we perform the orthonormalization process starting with the longest edge and then orthogonalizing the edges with successively decreasing lengths.

The following picture illustrates this situation for a parallel-epiped in the plane. There the edges of the parallel-epiped P (dotted lines) which is spanned by p_1, p_2 are orthogonalized in two different orders leaving first the shorter direction p_1 first to obtain a rectangle Q spanned by q_1, q_2 and then, second, leaving the longer direction p_2 fixed obtaining a rectangle Q' spanned by q_1', q_2'. Both rectangles enclose the parallel-epiped P but Q' is a much better enclosure than Q since the direction of the longest edge q_2' of Q' coincides with that of the longest edge p_2 of P.



Keep p_1, make p_2 orthogonal to p_1



Keep p_2, make p_1 orthogonal to p_2

This results in the following modified rule for the choice of B_{i+1}

$$B_{i+1} := Q_{i+1}, \quad \text{where} \quad \tilde{B}_{i+1} = \mathrm{mid}(\Diamond A_i B_i), \quad \tilde{B}_{i+1} P_{i+1} \approx Q_{i+1} R_{i+1} \ . \tag{2.33}$$

and P_{i+1} is the permutation matrix which reorders the columns of $\tilde{B}_{i+1}$ in the way described above.

Putting together the pieces, we now have a method for enclosing the solution $y_i, i \geq 0$ of a vector difference equation of the form (2.21):

Algorithm for first order vector difference equations:

1. compute an approximation $\tilde{y}_{i+1}$ (as a vector of floating-point numbers), (2.25).
2. compute an enclosure $[\hat{y}_{i+1}]$ of the error $\hat{y}_{i+1}$ by use of (2.27), choose B_{i+1} according to (2.33) and compute an enclosure of B_{i+1}^{-1} as in (2.31).

Obviously, this algorithm can even be applied if already the starting vector is an interval vector $y_0 = [y_0]$. Then we set $\tilde{y}_0 \approx \mathrm{mid}([y_0])$ and $[z_0] = [y_0] - \tilde{y}_0$ (and of course $B_0 = I$).

Now we turn to the case of scalar difference equations. We call the following recursive equations

$$\begin{aligned} &x_0, \ldots, x_{m-1} \ \text{ given} \\ &a_{i,m}x_{i+m} + \cdots + a_{i,0}x_i = b_i, \quad i \geq 0, \end{aligned} \tag{2.34}$$

a (scalar) linear difference equation of order m. The m starting values $x_0, \ldots, x_{m-1}$ are given and if we assume all $a_{i,m} \neq 0$ then the solution $x_m, x_{m+1}, \ldots$ can be computed recursively by

$$x_{i+m} = \frac{b_i}{a_{i,m}} - \frac{a_{i,0}}{a_{i,m}} x_i - \cdots - \frac{a_{i,m-1}}{a_{i,m}} x_{i+m-1} \ . \tag{2.35}$$

As we saw in Example 2.3 in the previous section, if we use interval arithmetic a recursively computed solution can yield rapidly growing intervals due to the wrapping effect. From Example 2.1 which is a first order difference equation for a matrix-vector equation, we know a geometric interpretation of this effect which will enable us to modify the computation in order to reduce the wrapping.

To this end we rewrite (2.34) as a first order matrix-vector equation by introducing the vectors

$$y_i := (x_i, \ldots, x_{i+m-1})^T \in \mathbb{R}^m, \quad i \geq 0, \tag{2.36}$$

$$d_i := (0, \ldots, 0, b_i/a_{i,m})^T \in \mathbb{R}^m, \quad i \geq 0, \tag{2.37}$$

and the matrices

$$A_i := \begin{pmatrix} 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & \ddots & \vdots & 0 \\ 0 & 0 & & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \\ \dfrac{-a_{i,0}}{a_{i,m}} & \dfrac{-a_{i,1}}{a_{i,m}} & \cdots & \dfrac{-a_{i,m-2}}{a_{i,m}} & \dfrac{-a_{i,m-1}}{a_{i,m}} \end{pmatrix} \in \mathbb{R}^{m \times m} \ . \tag{2.38}$$

Then the scalar m-th order equation (2.34) can be rewritten as the first order vector difference equation

$$y_{i+1} = A_i y_i + d_i, \quad i \geq 0 \tag{2.39}$$

with

$$y_0 = (x_0, \ldots, x_{m-1})^T \ , \tag{2.40}$$

which is exactly of the form (2.21).

Computing (2.39) directly in interval arithmetic is equivalent to the naive forward computation of (2.34) in interval arithmetic from which the reason of the severe overestimations becomes clear now also for scalar difference equations: they are just a special case of general first order vector difference equations and thus also subject to the wrapping effect.

Therefore we compute the solution $x_m, x_{m+1}, \ldots$ of (2.34) by using the equivalent vector form (2.39) and applying the algorithm discussed above for such equations.

2.2.2.2 Algorithms

We start this section by first listing some small auxiliary algorithms which will be needed later in the enclosure algorithm for the solution of difference equations.

The following algorithm sorts the columns of the input matrix B according to the lengths of the edges of the parallel-epiped $P(B, z)$ where the interval vector z is the second input parameter. The procedure SWAP used here swaps the contents of its two parameters.

Algorithm 2.7: SortColumns (B, z) {procedure}

{Sort the columns of B by decreasing length of the edges of P(B,z)}
input: *rmatrix* B, interval vector z
output: *rmatrix* B with columns rearranged

(a) {compute lengths of edges (squares of these lengths suffice):}
for $i := 1$ **to** m **do** $length_i := (B_{*,i} \cdot B_{*,i}) \cdot \mathrm{diam}(z_i)^2$

(b) {sort columns of B according to these lengths}
for $i := 1$ **to** $m - 1$ **do**
for $j := i + 1$ **to** m **do**
    if $length_i < length_j$ **then** Swap $(B_{*,i}, B_{*,j})$
        Swap $(length_i, length_j)$

(c) **return** B

Next an algorithm is presented which computes an approximate QR-decomposition of the input matrix A in ordinary floating-point arithmetic. We did not present the theory for this method in the previous section since this is a standard algorithm, see e.g. [309]. The matrix A is successively multiplied with Householder matrices to bring it in triangular form R. The orthogonal part Q is then the accumulated product of the Housholder matrices.

Algorithm 2.8: QR (A) {function}

{floating-point computation of the QR-decomposition of A }
input: *rmatrix* A
output: almost orthogonal matrix Q (function result)

(a) {start with identity matrix}
$Q := I$

(b) {loop over all columns of A (but the last)}
for $k := 1$ **to** $m-1$ **do**
$b := A_{k..n,k}$ { b is $(n-k+1)$-vector }
if $b \neq 0$ **then**
$s := -\text{sign}(A_{k,k}) \cdot \|b\|_2$
$b_k := A_{k,k} - s$
$s := 1/(s \cdot b_k)$
{eliminate entries below diagonal in A:}
for $i := k+1$ **to** n **do**
$r := s \cdot (b \cdot A_{k..n,i})$
for $j := k$ **to** n **do** $A_{j,i} := A_{j,i} + b_j \cdot r$
{accumulate product of Householder matrices in Q:}
for $i := 1$ **to** n **do**
$r := s \cdot (Q_{i,k..n} \cdot b)$
for $j := k$ **to** n **do** $Q_{i,j} := Q_{i,j} + b_j \cdot r$

(c) {return function value}
$\mathsf{QR} := Q$

The next algorithm is a function which computes an enclosure of the inverse of an almost orthogonal matrix Q using the estimates of the Neumann series. An error code 1 is returned if the norm $q = ||I - Q^T Q||_\infty \geq 1$, otherwise an error code 0 is returned.

Algorithm 2.9: Inv (Q, err_code) {function}

{Enclosure of the inverse of an almost orthogonal matrix Q }
input: *rmatrix* Q (almost orthogonal)
output: Enclosure of Q^{-1} as *imatrix* (function result), *integer* err_code indicating successful inversion (=0 or =1 otherwise).

(a) {norm of residuum}
$q := \|I - Q^T Q\|_\infty$
if $q \geq 1$ **then** $err\_code := 1$ **else** $err\_code := 0$
if $err\_code = 1$ **then return**

(b) $\epsilon := \|Q^T\|_\infty \cdot q/(1-q)$
for $i := 1$ **to** m **do**
for $j := 1$ **to** m **do**
$Q1_{i,j} := Q_{j,i} + [-\epsilon, \epsilon]\|Q^T\|_\infty$

(c) {return enclosure of inverse as function value:}
$\mathsf{Inv} := Q1$

The following algorithm computes an enclosure of the solution of a first order vector difference equation using the coordinate transformation with orthogonal basis matrices as discussed in the previous section. In the algorithm only one step is computed, the input for the algorithm is a parallel-epidped P_0(y0,B0,z0) and output is a parallel-epiped P_1(y1,B1,z1). To obtain a component-wise interval enclosure of P_1 the interval vector y1 + B1*z1 must be computed after a call to this algorithm.

The input matrix A (=A_i) and the input vector d (=d_i) are chosen as interval vectors here to allow for interval data and for non floating-point data (enclosed in intervals).

Algorithm 2.10: MatVecIter $(A, d, y0, B0, z0, y1, B1, z1)$ {procedure}

{Compute one step of the matrix-vector iteration $y_{i+1} = Ay_i + d_i$ }
input: *imatrix* $A = (A_i)$, *rmatrix* $B0 = (B_i)$ (from previous step), *ivector* $d = (d_i)$, *rvector* $y0 = (\tilde{y}_i)$ and *ivector* $z0 = ([z_i])$ (from previous step).
output: *rvector* $y1 = (\tilde{y}_{i+1})$, *rmatrix* $B1 = (B_{i+1})$ and *ivector* $z1 = ([z_{i+1}])$.

(a) {compute approximation}
$y1 := \text{mid}(A) \cdot y0 + \text{mid}(d)$

(b) {enclosure of the error}
$B1 := \text{mid}(A) \cdot B0$
SortColumns $(B1, z0)$
QR $(B1)$
$BI1 :=$ Inv $(B1, err\_code)$
if $err\_code = 1$ **then** print warning: inversion not successful
$z1 := (BI1 \cdot A \cdot B0) \cdot z0 + BI1 \cdot \Diamond(d + A \cdot y0 - y1)$

(c) **return** $y1, B1, z1$

Finally we list an algorithm for the computation of the solution of a scalar difference equation of order m. It is essentially identical with the previous algorithm 2.2.10 for first order vector difference equations, only the form of the input is different, i.e. here we only need an ivector a, containing the coefficients $a_0, \ldots, a_m$ and an interval b for b_i.

In this algorithm we use a pseudo-function `COEFF` which should just be an indication than its argument should not be interpreted as a vector but rather as a matrix of the form (2.38) containing the coefficients of the argument vector in the way indicated in (2.38).

Algorithm 2.11: DiffIter $(A, b, y0, B0, z0, y1, B1, z1)$ {procedure}

{Compute one step of the scalar iteration with the difference equation $a_{i,m}x_{i+m} + \cdots + a_{i,0}x_i = b_i$ }
input: *ivector* A (coefficient vector), *interval* b (right hand side), *rmatrix* $B0 = (B_i)$ (from previous step), *rvector* $y0 = (\tilde{y}_i)$ and *ivector* $z0 = ([z_i])$ (from previous step).
output: *rvector* $y1 = (\tilde{y}_{i+1})$, *rmatrix* $B1 = (B_{i+1})$ and *ivector* $z1 = ([z_{i+1}])$.

(a) {compute floating-point approximation}
$y1_{1..m-1} := y0_{2..m}$
$y1_m := (\text{mid}(b) - \sum_{i=1}^{m} \text{mid}(a_{i-1}) \cdot y0_i)/\text{mid}(a_m)$

(b) {enclosure of the error}
$B1 :=$ Coeff $(\text{mid}(A)) \cdot B0$
SortColumns $(B1, z0)$
QR $(B1)$

$BI1 := \mathsf{Inv}\,(B1)$
$d := \Diamond(b - \sum_{i=1}^{m} a_{i-1} \cdot y0_i)/a_m - y1_m$
$z1 := (BI1 \cdot (\mathsf{Coeff}\,(A) \cdot B0)) \cdot z0 + BI1_{*,m} \cdot d$

(c) **return** $y1, B1, z1$

2.2.2.3 PASCAL-XSC Program Code

The following PASCAL-XSC module contains the algorithms from the previous section and exports the last two algorithms `Mat_Vec_Iter` and `Diff_Iter` as global entries.

```
MODULE mv_differ;

USE i_ari,mv_ari,mvi_ari;

FUNCTION max( a,b : REAL ) : REAL;
BEGIN
   IF a>b THEN max:= a ELSE max:= b;
END;

FUNCTION max_norm( VAR a : RVECTOR ) : REAL;
{ upper bound for max-norm of rvector a }
VAR i : INTEGER;
    s : REAL;
BEGIN
   s:= 0;
   FOR i:= lb(a) TO ub(a) DO s:= s +> abs(a[i]);
   max_norm:= s;
END;

FUNCTION max_norm( VAR a : RMATRIX ) : REAL;
{ upper bound for row sum norm of rmatrix a }
VAR i : INTEGER;
    s : REAL;
BEGIN
   s:= 0;
   FOR i:= lb(a) TO ub(a) DO s:= max( s, max_norm(a[i]) );
   max_norm:= s;
END;

FUNCTION max_norm( VAR a : IVECTOR ) : REAL;
{ upper bound for max-norm of ivector a }
VAR i : INTEGER;
    s : REAL;
BEGIN
   s:= 0;
   FOR i:= lb(a) TO ub(a) DO s:= s +> sup(abs(a[i]));
   max_norm:= s;
END;

FUNCTION max_norm( VAR a : IMATRIX ) : REAL;
{ upper bound for row sum norm of imatrix a }
VAR i : INTEGER;
    s : REAL;
BEGIN
   s:= 0;
   FOR i:= lb(a) TO ub(a) DO s:= max( s, max_norm(a[i]) );
   max_norm:= s;
END;
```

```
PROCEDURE inv( VAR q : RMATRIX; VAR q_inv : IMATRIX;
                                VAR err_code : INTEGER );
{ compute enclosure for inverse of an almost orthogonal matrix q }
VAR nrm : REAL;
    det : interval;
    i,j : INTEGER;
    qt  : RMATRIX[lb(q,1)..ub(q,1),lb(q,2)..ub(q,2)];
BEGIN
   err_code:= 0;
   IF ub(q,1)-lb(q,1) = 0 THEN { 1x1 - matrix }
      q_inv:= intval(1.0)/q[lb(q,1),lb(q,2)] ELSE
   IF ub(q,1)-lb(q,1) = 1 THEN { 2x2 - matrix }
   BEGIN
      det:= ##(   q[lb(q,1),lb(q,2)]*q[ub(q,1),ub(q,2)]
                - q[lb(q,1),ub(q,2)]*q[ub(q,1),lb(q,2)] );
      IF 0.0 in det THEN err_code:= 1 ELSE
      BEGIN
         q_inv[lb(q_inv,1),lb(q_inv,2)]:=  q[ub(q,1),ub(q,2)] / det;
         q_inv[lb(q_inv,1),ub(q_inv,2)]:= -q[lb(q,1),ub(q,2)] / det;
         q_inv[ub(q_inv,1),lb(q_inv,2)]:= -q[ub(q,1),lb(q,2)] / det;
         q_inv[ub(q_inv,1),ub(q_inv,2)]:=  q[lb(q,1),lb(q,2)] / det;
      END;
   END ELSE
   BEGIN
      qt:= transp(q);
      nrm:= max_norm( ##(id(q)-qt*q) );
      IF nrm < 1.0 THEN
      BEGIN
         nrm:= max_norm(qt) *> nrm /> ( 1.0 -< nrm );
         det:= intval(-nrm,nrm);
         FOR i:= lb(q,1) TO ub(q,1) DO
         FOR j:= lb(q,2) TO ub(q,2) DO  q_inv[i,j]:= qt[i,j] + det;
      END ELSE
      BEGIN
         err_code:= 1;
         q_inv:= qt;
      END;
   END;
END;

FUNCTION QR ( VAR A : RMATRIX ) : RMATRIX[1..ub(A),1..ub(A)];
VAR i,j,k,n : INTEGER;
    Q       : RMATRIX [1..ub(A),1..ub(A)];
    r,s     : REAL;
    b,c,d   : RVECTOR [1..ub(A)];
BEGIN
   n:= ub(A);
   Q:= id(Q);
   FOR k:= 1 TO n-1 DO
   BEGIN
      b[k..n]:= A[k..n,k];
      IF RVECTOR(b[k..n]) <> null(b[k..n]) THEN
      BEGIN
         s:= -sign(A[k,k])*sqrt(b*b);
         b[k]:= A[k,k] - s;
         s:= 1 / (s * b[k]);
         FOR i:= k+1 TO n DO
         BEGIN
            r:= s * #*( FOR j:= k TO n SUM ( b[j]*A[j,i] ) );
            FOR j:= k TO n DO A[j,i]:= A[j,i] + b[j] * r;
         END;
         FOR i:= 1 TO n DO
         BEGIN
            r:= s * #*( FOR j:= k TO n SUM ( b[j]*Q[i,j] ) );
```

```
            FOR j:= k TO n DO Q[i,j]:= Q[i,j] + b[j] * r;
         END;
      END;
      b[k]:= 0;
   END;
   QR:= Q;
END;

PROCEDURE sort_columns( VAR B : RMATRIX; VAR z : IVECTOR );
VAR i,j,m    : INTEGER;
    hv,length : RVECTOR[1..ub(B)];
    hr        : REAL;
BEGIN
   m:= ub(B);
   FOR i:= 1 TO m DO
       length[i]:= sqr(diam(z[i])) * (RVECTOR(B[*,i])*RVECTOR(B[*,i])) ;
   FOR i:=  1  TO m-1 DO
   FOR j:= i+1 TO  m  DO
       IF length[i] < length[j] THEN
       BEGIN
          hr:= length[j];       length[j]:= length[i]; length[i]:= hr;
          hv:= RVECTOR(B[*,j]);    B[*,j]:= B[*,i];       B[*,i]:= hv;
       END;
END;

GLOBAL PROCEDURE
          mat_vec_iter( VAR  A : IMATRIX; VAR  d : IVECTOR;
                        VAR B0 : RMATRIX; VAR z0 : IVECTOR; VAR y0 : RVECTOR;
                        VAR B1 : RMATRIX; VAR z1 : IVECTOR; VAR y1 : RVECTOR;
                        VAR err_code : INTEGER );

{ Compute one step of the matrix-vector iteration x[n+1] = A*x[n] + d        }
{ using the input parallel-epiped P(B0,z0,y0) containing x[n] and giving as  }
{ result the output parallel-epiped P(B1,z1,y1) containing x[n+1].           }
{ Attention: all parameters must be different variables in the calling       }
{ program !                                                                  }

VAR B1_inv : IMATRIX[lb(A,1)..ub(A,1),lb(A,2)..ub(A,2)];
BEGIN
   B1:= mid(A);
   { floating-point approximation : }
   y1:= B1*y0 + mid(d);
   { enclosure of error of y1 : }
   B1:= B1*B0;
   sort_columns( B1, z0 );
   B1:= QR( B1 );
   inv( B1,B1_inv, err_code );
   IF err_code<>0 THEN
      writeln( 'Error in matrix inversion, err_code = ',err_code );
   z1:= ((B1_inv*A)*B0)*z0 + B1_inv*(d + A*y0 - y1 );
END;

GLOBAL PROCEDURE
             diff_iter( VAR  a : IVECTOR; VAR  b : INTERVAL;
                        VAR B0 : RMATRIX; VAR z0 : IVECTOR; VAR y0 : RVECTOR;
                        VAR B1 : RMATRIX; VAR z1 : IVECTOR; VAR y1 : RVECTOR;
                        VAR err_code : INTEGER );

{ Compute one step of the scalar iteration                                   }
{            a[m]*x[i+m] + ... + a[0]*x[i] = b                               }
{ by an equivalent matrix-vector iteration using the input parallel-epiped   }
{          P(B0,z0,y0) containing (x[i],...x[i+m-1])                         }
{ and giving as result the output parallel-epiped                            }
{          P(B1,z1,y1) containing (x[i+1],...,x[i+m])                        }
```

```
{ The final value x[i+m] of this iteration is then contained in the last      }
{ component of the interval vector y1 + B1*z1                                 }
{ All index ranges are assumed to start at 1, except for A which starts at 0!}
{ Attention: all parameters must be different variables in the calling       }
{ program !                                                                   }

TYPE coeff_rmatrix = RVECTOR;
     coeff_imatrix = IVECTOR;

OPERATOR * ( VAR A : coeff_rmatrix; VAR B : RMATRIX )
           frm_rm_mul : RMATRIX[1..ub(B),1..ub(B)];
VAR i,j : INTEGER;
BEGIN
   FOR i:= 1 TO ub(B)-1 DO frm_rm_mul[i]:= B[i+1];
   FOR j:= 1 TO  ub(B)  DO frm_rm_mul[ub(B),j]:=
      - #*( FOR i:= 1 TO ub(B) SUM ( A[i-1] * B[i,j] ) ) / A[ub(B)];
END;

OPERATOR * ( VAR A : coeff_imatrix; VAR B : RMATRIX )
           fim_rm_mul : IMATRIX[1..ub(B),1..ub(B)];
VAR i,j : INTEGER;
BEGIN
   FOR i:= 1 TO ub(B)-1 DO fim_rm_mul[i]:= B[i+1];
   FOR j:= 1 TO  ub(B)  DO fim_rm_mul[ub(B),j]:=
      - ##( FOR i:= 1 TO ub(B) SUM ( A[i-1] * B[i,j] ) ) / A[ub(B)];
END;

VAR i,m    : INTEGER;
    bm     : REAL;
    d      : INTERVAL;
    Am     : coeff_rmatrix[0..ub(A)];
    B1_inv : IMATRIX[1..ub(A),1..ub(A)];
BEGIN
   m:= ub(A);
   bm:= mid(b);
   Am:= coeff_rmatrix( mid(A) );
   { floating-point approximation : }
   FOR i:= 1 TO m-1 DO y1[i]:= y0[i+1];
   y1[m]:= #*( bm - FOR i:= 1 TO m SUM ( Am[i-1]*y0[i] ) ) / Am[m];
   { enclosure of error of y1 : }
   B1:= Am*B0;
   sort_columns( B1, z0 );
   B1:= QR( B1 );
   inv( B1,B1_inv, err_code );
   IF err_code<>0 THEN
      writeln( 'Error in matrix inversion, err_code = ',err_code );
   d:= ##( b - FOR i:= 1 TO m SUM ( A[i-1]*y0[i] ) ) / a[m] - y1[m];
   z1:= (B1_inv*(coeff_imatrix(A)*B0))*z0 + B1_inv[*,m]*d;
END;


END.
```

2.2.2.4 Test Results

To test the first order matrix-vector difference equations we use the following simple iteration which is a two-dimensional rotation around the origin by an angle ϕ with $\cos\phi = 0.6$ and $\sin\phi = 0.8$.

$$y_{i+1} = \begin{pmatrix} 0.6 & 0.8 \\ -0.8 & 0.6 \end{pmatrix} y_i \quad , \quad i \geq 0, \quad y_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} . \tag{2.41}$$

The following short program is written for the simple case of A_i and d_i being independent of i. It reads the dimension n the matrix A, the vector d the initial vector y_0 and the number of iterations to be performed. It prints out the results obtained by applying parallel-epipeds with orthogonal basis matrix as described in (2.33) and the results computed without any basis transformation i.e. by forward computation in plain interval arithmetic ($B_i = I$ for alls i).

```
program mv_iteration;

USE i_ari,mv_ari,mvi_ari,mv_differ;

PROCEDURE main( m : INTEGER );
VAR i,n,err_code  : INTEGER;
    A             : imatrix[1..m,1..m];
    B0,B1         : rmatrix[1..m,1..m];
    y0,y1         : rvector[1..m];
    z0,z1,x0,x1,d : ivector[1..m];
BEGIN
   WRITE('A  = '); READ(A);
   WRITE('d  = '); READ(d);
   REPEAT
      WRITE('x0 = '); READ(x0);
      WRITE('n  = '); READ(n);

      B0:= id(B0);
      y0:= mid(x0);
      z0:= x0 - y0;

      FOR i:= 1 TO n DO
      BEGIN
         mat_vec_iter( A,d, B0,z0,y0, B1,z1,y1, err_code );
         x1:= A*x0 + d;
         IF err_code <> 0 THEN WRITELN('Error in iteration ',i );
         WRITELN('i = ',i:3);
         WRITE('orthogonal basis Q :'); WRITE( y1 + B1*z1 );
         WRITE('no transformation  :'); WRITELN( x1 );
         B0:= B1;
         z0:= z1;
         y0:= y1;
         x0:= x1;
      END;
   UNTIL n <= 0;
END;

VAR m : INTEGER;

BEGIN
   REPEAT
      WRITE('m = '); READ(m);
      IF m > 0 THEN main(m);
   UNTIL m <= 0;
END.
```

The result for our small test system is listed next, where, however most of the output is deleted and only the result for a few selected values of i between 1 and 1000 are printed. Obviously the iteration with the orthogonal basis transformation can still be computed over a far wider range of values of i whereas the straigthforward interval computation is hopelessly bad.

```
m = 2
A  = 0.6 0.8 -0.8 0.6
d  = 0 0
x0 = 1 0
n  = 1000
i =   1
orthogonal basis Q :
[  5.999999999999998E-001,  6.000000000000002E-001 ]
[ -8.000000000000002E-001, -7.999999999999998E-001 ]
no transformation  :
[  5.999999999999999E-001,  6.000000000000001E-001 ]
[ -8.000000000000001E-001, -7.999999999999999E-001 ]

i =   2
orthogonal basis Q :
[  -2.80000000000001E-001,  -2.79999999999999E-001 ]
[ -9.600000000000005E-001, -9.599999999999996E-001 ]
no transformation  :
[ -2.800000000000002E-001, -2.799999999999997E-001 ]
[ -9.600000000000003E-001, -9.599999999999998E-001 ]

i =   3
orthogonal basis Q :
[  -9.36000000000001E-001,  -9.35999999999999E-001 ]
[  -3.52000000000001E-001,  -3.51999999999999E-001 ]
no transformation  :
[  -9.36000000000001E-001,  -9.35999999999999E-001 ]
[  -3.52000000000001E-001,  -3.51999999999999E-001 ]
    .
    .
    .
i =  10
orthogonal basis Q :
[  -9.88496588800002E-001,  -9.88496588799998E-001 ]
[   -1.5124316160001E-001,   -1.5124316159999E-001 ]
no transformation  :
[   -9.8849658880001E-001,   -9.8849658879999E-001 ]
[    -1.512431616001E-001,    -1.512431615999E-001 ]
    .
    .
    .
i =  50
orthogonal basis Q :
[   -7.2543585253528E-001,   -7.2543585253525E-001 ]
[   -6.8828978189165E-001,   -6.8828978189162E-001 ]
no transformation  :
[         -7.2543586E-001,         -7.2543584E-001 ]
[         -6.8828979E-001,         -6.8828977E-001 ]
    .
    .
    .
i = 100
orthogonal basis Q :
[     5.251435228713E-002,     5.251435228716E-002 ]
[    9.9862016943572E-001,    9.9862016943576E-001 ]
no transformation  :
[               -5.3E-002,                1.6E-001 ]
[                8.9E-001,                1.2E+000 ]
    .
    .
    .
i = 200
orthogonal basis Q :
[    -9.944844856078E-001,    -9.944844856076E-001 ]
```

```
[     1.048837827575E-001,     1.048837827577E-001 ]
no transformation  :
[                -4.4E+013,                4.4E+013 ]
[                -4.4E+013,                4.4E+013 ]
    .
    .
    .
i = 500
orthogonal basis Q :
[     2.596817149124E-001,     2.596817149127E-001 ]
[     9.656942616273E-001,     9.656942616275E-001 ]
no transformation  :
[                -3.0E+057,                3.0E+057 ]
[                -3.0E+057,                3.0E+057 ]
    .
    .
    .
i = 1000
orthogonal basis Q :
[    -8.651308138804E-001,    -8.651308138799E-001 ]
[     5.015462838810E-001,     5.015462838815E-001 ]
no transformation  :
[                -3.5E+130,                3.5E+130 ]
[                -3.5E+130,                3.5E+130 ]
```

2.2.3 Band Matrices

Matrices with band structure and difference equations are closely related. We have seen this already in the examples of Section 2.2.1. There the difference equations could be rewritten equivalently as a linear system with band matrix which was triangular even. Similarly we can go in the other direction and write a triangular banded matrix as a difference equation.

The system

$$Ax = \begin{pmatrix} a_{1,1} & & & & \\ \vdots & \ddots & & 0 & \\ a_{m,1} & & \ddots & & \\ & \ddots & & \ddots & \\ 0 & & a_{n,n-m+1} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \\ b_n \end{pmatrix} = b \tag{2.42}$$

is obviously equivalent with the difference equation

$$a_{i,i-m+1}x_{i-m+1} + \cdots + a_{i,i}x_i = b_i \ , \quad i = m, \ldots, n, \tag{2.43}$$

of order $m-1$ with the starting values

$$x_i = (b_i - a_{i,i-1}x_{i-1} - \cdots - a_{i,1}x_1)/a_{i,i} \ , \quad i = 1, \ldots, m-1 \ . \tag{2.44}$$

In the examples in Section 2.2.1 we saw that the solution of triangular systems by interval forward substitution can result in severe overestimations. Therefore we use the relationship to difference equations and solve triangular banded systems with the method for difference equations which was presented in the previous Section.

For general banded systems we will then apply a LU-decomposition without pivoting to the coefficient matrix A and derive an interval iteration similar to (2.5). Here, however, we will not use a full approximate inverse R but rather the iteration will be performed by solving two systems with banded triangular matrices (L and U).

A similar approach to banded and sparse linear systems can be found in [78], [79] and [295]. There, however, the triangular systems were solved by interval forward and backward substitution which often results in gross overestimations as we have seen already.

For a different approach to the verified solution of linear systems with large banded or arbitrary sparse coefficient matrix see Rump, [292].

2.2.3.1 Theoretical Background

The mathematical background for the verified solution of large linear systems with band matrices is exactly the same as it was already in Section 2.1.2 for systems with dense matrices. To avoid repetition of this section we assume the same notation here.

For dense systems the interval iteration (2.5) was derived by use of an approximate inverse R of the coefficient matrix A. This is however what we want to avoid for large banded matrices A. Therefore we chose a different approximate inverse, namely

$$R := (LU)^{-1} \approx A^{-1} \tag{2.45}$$

where

$$LU \approx A \tag{2.46}$$

is an approximate LU-decomposition of A without pivoting. Since we do not use pivoting both L and U are banded matrices again, and of course they are lower and upper triangular, resp.

The analogue of the iteration (2.5) now reads in our case

$$y_{k+1} = (LU)^{-1}(b - A\tilde{x}) + (I - (LU)^{-1}A)y_k \tag{2.47}$$

or, multiplying with LU and taking intervals:

$$LU[y]_{k+1} = \Diamond(b - A\tilde{x}) + \Diamond(LU - A)[y]_k. \tag{2.48}$$

Therefore we have to solve two linear systems with triangular, banded coefficient matrices, L and U, in order to compute $[y]_{k+1}$, i.e. to perform one step of the iteration (2.48).

In each iteration we first compute an enclosure for the solution of

$$L[z]_{k+1} = \Diamond(b - A\tilde{x}) + \Diamond(LU - A)[y]_k$$

and then $[y]_{k+1}$ from

$$U[y]_{k+1} = [z]_{k+1}.$$

In both systems we do not use just plain interval forward or backward substitution, however, as discussed above, we treat the systems as difference equation and apply the corresponding method from the previous Section.

Here again, as in Section 2.1.2 the inclusion test

$$[y]_{k+1} = F([y]_k) \subset [y]_k^\circ \tag{2.49}$$

has to be checked in the same way and if it is satisfied then the same assertions hold as in the dense case.

Remark:

If we compute the LU-decomposition with Crout's algorithm, then we can get the matrix $\Diamond(LU - A)$ virtually for free, since the scalar products which are needed here have to be computed in Crout's algorithm anyway.

2.2.3.2 PASCAL-XSC Program Code

The following program implements the method from the last Subsection for the verified solution of linear systems with band matrices. In addition to the implementation of the solution method the program includes also a small demonstration part which can be used to solve some simple systems.

First the program reads the number of lower and upper bands and then one value for each of the bands, i.e. initially a Toeplitz matrix is generated.

In the next step, however, any number of elements of the matrix can be changed, such that also arbitrary band matrices can be entered. To change the element $a_{i,j}$ only i, j and the new value for this element must be entered. Changing elements is finished by entering zeros for i and j.

Next the right hand side must be entered. There are several choices of predefined solutions, such that the right hand side b will be determined from this given solution. Alternatively b can be set to a constant value in all components or all components can be entered successively. In any case, the values of the components of b may be changes again similarly as for the matrix. When no changes are done anymore, the solution algorithm starts.

At the end the solution and a brief error statistics is printed out (see next Subsection).

```
PROGRAM band;

USE i_ari,mv_ari,mvi_ari;

VAR k,l,n : integer;

FUNCTION min( a,b : integer ) : integer;
BEGIN
   IF a<b THEN min:= a ELSE min:= b;
END;

FUNCTION max( a,b : integer ) : integer;
BEGIN
   IF a>b THEN max:= a ELSE max:= b;
END;
```

```
FUNCTION max( a,b,c : integer ) : integer;
BEGIN
   IF a>b THEN max:= max(a,c) ELSE max:= max(b,c);
END;

FUNCTION max( a,b : real ) : real;
BEGIN
   IF a>b THEN max:= a ELSE max:= b;
END;

FUNCTION norm( VAR a : rmatrix ) : real;
{ compute row-sum norm of a }
VAR i,j : integer;
    m,s : real;
    v   : rvector[lb(a,2)..ub(a,2)];
BEGIN
   m:= 0.0;
   FOR i:= lb(a,1) TO ub(a,1) DO
   BEGIN
      FOR j:= lb(a,2) TO ub(a,2) DO v[j]:= abs(a[i,j]);
      s:= #>( FOR j:= lb(a,2) TO ub(a,2) SUM ( v[j] ) );
      m:= max(m,s);
   END;
   norm:= m;
END;

FUNCTION norm( VAR a : imatrix ) : real;
{ compute row-sum norm of a }
VAR i,j : integer;
    m,s : real;
    v   : rvector[lb(a,2)..ub(a,2)];
BEGIN
   m:= 0.0;
   FOR i:= lb(a,1) TO ub(a,1) DO
   BEGIN
      FOR j:= lb(a,2) TO ub(a,2) DO v[j]:= sup(abs(a[i,j]));
      s:= #>( FOR j:= lb(a,2) TO ub(a,2) SUM ( v[j] ) );
      m:= max(m,s);
   END;
   norm:= m;
END;

PROCEDURE inv( VAR a : rmatrix; VAR a_inv : imatrix; VAR err : integer );
{ compute enclosure of the inverse of an almost orthogonal matrix a }
VAR nrm   : real;
    delta : interval;
    i,j   : integer;
    at    : rmatrix[lb(a,1)..ub(a,1),lb(a,2)..ub(a,2)];
BEGIN
   err:= 0;
   IF ub(a,1)-lb(a,1) = 0 THEN a_inv:= intval(1.0)/a[lb(a,1),lb(a,2)] ELSE
   IF ub(a,1)-lb(a,1) = 1 THEN { 2x2 - matrix }
   BEGIN
      delta:= ##(   a[lb(a,1),lb(a,2)]*a[ub(a,1),ub(a,2)]
                  - a[lb(a,1),ub(a,2)]*a[ub(a,1),lb(a,2)] );
      IF 0.0 IN delta THEN err:= 1 ELSE
      BEGIN
         a_inv[lb(a_inv,1),lb(a_inv,2)]:=  a[ub(a,1),ub(a,2)] / delta;
         a_inv[lb(a_inv,1),ub(a_inv,2)]:= -a[lb(a,1),ub(a,2)] / delta;
         a_inv[ub(a_inv,1),lb(a_inv,2)]:= -a[ub(a,1),lb(a,2)] / delta;
         a_inv[ub(a_inv,1),ub(a_inv,2)]:=  a[lb(a,1),lb(a,2)] / delta;
      END;
   END ELSE
```

```
    BEGIN
       at:= transp(a);
       nrm:= norm( ##(id(a)-at*a) );
       IF nrm < 1.0 THEN
       BEGIN
          nrm:= norm(at) *> nrm /> ( 1.0 -< nrm );
          delta:= intval(-nrm,nrm);
          FOR i:= lb(a,1) TO ub(a,1) DO
          FOR j:= lb(a,2) TO ub(a,2) DO  a_inv[i,j]:= at[i,j] + delta;
       END ELSE
       BEGIN
          err:= 1;
          a_inv:= at;
       END;
    END;
END;

FUNCTION QR ( VAR A : RMATRIX ) : RMATRIX[1..ub(A),1..ub(A)];
{ perform QR-decomposition of A by use of Householder matrices }
VAR i,j,k,n : INTEGER;
    Q       : RMATRIX [1..ub(A),1..ub(A)];
    r,s     : REAL;
    b,c,d   : RVECTOR [1..ub(A)];
BEGIN
   n:= ub(A);
   Q:= id(Q);
   FOR k:= 1 TO n-1 DO
   BEGIN
      b[k..n]:= A[k..n,k];
      IF RVECTOR(b[k..n]) <> null(b[k..n]) THEN
      BEGIN
         s:= sqrt(b*b);
         IF A[k,k] >= 0.0 THEN s:= -s;
         b[k]:= A[k,k] - s;
         s:= 1 / (s * b[k]);
         FOR i:= k+1 TO n DO
         BEGIN
            r:= s * #*( FOR j:= k TO n SUM ( b[j]*A[j,i] ) );
            FOR j:= k TO n DO A[j,i]:= A[j,i] + b[j] * r;
         END;
         FOR i:= 1 TO n DO
         BEGIN
            r:= s * #*( FOR j:= k TO n SUM ( b[j]*Q[i,j] ) );
            FOR j:= k TO n DO Q[i,j]:= Q[i,j] + b[j] * r;
         END;
      END;
      b[k]:= 0;
   END;
   QR:= Q;
END;

FUNCTION QR( VAR A : rmatrix; VAR y : ivector ) : rmatrix[1..ub(A),1..ub(A)];
{ sort columns of A according to increasing length and then do a QR-decomp. }
VAR i,j,k,n   : integer;
    hv,laenge : rvector[1..ub(A)];
    hr        : real;
BEGIN
   n:= ub(A);
   FOR i:= 1 TO n DO
       laenge[i]:= sqr(diam(y[i])) * (rvector(A[*,i])*rvector(A[*,i])) ;
   FOR j:=  1  TO n-1 DO
   FOR k:= j+1 TO  n  DO
   BEGIN
      IF laenge[j] < laenge[k] THEN
```

```
      BEGIN
         hr:= laenge[k];        laenge[k]:= laenge[j]; laenge[j]:= hr;
         hv:= rvector(A[*,k]);    A[*,k]:= A[*,j];       A[*,j]:= hv;
      END;
   END;
   QR:= QR(A);
END;

PROCEDURE lss_triangular( VAR A : rmatrix; VAR b,x : rvector );

{ floating point forward substitution for lower triangular system Ax=b  or   }
{ floating point backward substitution for upper triangular system Ax=b .    }
{ A must be [1..n,-1..0] or [1..n,0..k]                                       }

VAR i,j,n,k,l : integer;
BEGIN
   n:=      ub(A,1);
   l:= abs( lb(A,2) );
   k:=      ub(A,2);
   IF k = 0 THEN    { A is  l o w e r  triangular matrix }
      FOR i:= 1 TO n DO
       x[i]:=
        #*( b[i] - FOR j:= max(1,i-l) TO i-1 SUM (A[i,j-i]*x[j]) ) / A[i,0]
   ELSE
   IF l = 0 THEN    { a is  u p p e r  triangular matrix }
      FOR i:= n DOWNTO 1 DO
       x[i]:=
        #*( b[i] - FOR j:= i+1 TO min(n,i+k) SUM (A[i,j-i]*x[j]) ) / A[i,0]
   ELSE
      { Error:  A is not a regular triangular matrix !  } ;
END;

TYPE coefficient_matrix = rvector;

OPERATOR * ( VAR a : coefficient_matrix; VAR b : rmatrix )
                       cmmulrm : imatrix[lb(b,1)..ub(b,1),lb(b,2)..ub(b,2)];
VAR i,j,n : integer;
BEGIN
   n:=ub(b);
   FOR i:= 1 TO n-1 DO cmmulrm[i]:= b[i+1];
   FOR j:= 1 TO n DO cmmulrm[n,j]:= ##( FOR i:= 1 TO n SUM (a[i]*b[i,j]) );
END;

PROCEDURE lss_lower( VAR A : rmatrix; VAR b,x : ivector; l : integer );
{ forward substitution using coordinate transformations }
VAR i,j,n,err: integer;
    af       : coefficient_matrix[1..l];
    c0,c1    : rmatrix[1..l,1..l];
    c_inv,ai : imatrix[1..l,1..l];
    y0,y1,bi : ivector[1..l];
BEGIN
   n:= ub(A,1);
   c0:= id(c0);
   bi:= 0.0;
   FOR i:= 1 TO l DO
   BEGIN
       x[i]:= ##( b[i] - FOR j:= 1 TO i-1 SUM (A[i,j-i]*x[j]) ) / A[i,0];
       y0[i]:= x[i];
   END;
   FOR i:= l+1 TO n DO
   BEGIN
      FOR j:= 1 TO l DO af[j]:= -A[i,j-l-1]/A[i,0];
      bi[l]:= b[i] / a[i,0];
      ai:= af*c0;
```

```
      c1:= QR( mid(ai), y0 );
      inv(c1,c_inv,err);
      IF err<>0 THEN
        writeln( 'Error in matrix inversion, err = ',err, '   i = ',i );
      y0:= (c_inv*ai)*y0 + c_inv*bi;
      x[i]:= c1[1]*y0;
      c0:= c1;
   END;
END;

PROCEDURE lss_upper( VAR A : rmatrix; VAR b,x : ivector; k : integer );
{ backward substitution using coordinate transformations }
VAR i,j,n,err: integer;
    af       : coefficient_matrix[1..k];
    c0,c1    : rmatrix[1..k,1..k];
    c_inv,ai : imatrix[1..k,1..k];
    y0,y1,bi : ivector[1..k];
BEGIN
   n:= ub(A,1);
   c0:= id(c0);
   bi:= 0.0;
   FOR i:= n DOWNTO n-k+1 DO
   BEGIN
       x[i]:= ##( b[i] - FOR j:= i+1 TO n SUM (A[i,j-i]*x[j]) ) / A[i,0];
       y0[n+1-i]:= x[i];
   END;
   FOR i:= n-k DOWNTO 1 DO
   BEGIN
      FOR j:= 1 TO k DO af[j]:= -A[i,k+1-j]/A[i,0];
      bi[k]:= b[i] / a[i,0];
      ai:= af*c0;
      c1:= QR( mid(ai), y0 );
      inv(c1,c_inv,err);
      IF err<>0 THEN
        writeln( 'Error in matrix inversion, err = ',err, '   i = ',i );
      y0:= (c_inv*ai)*y0 + c_inv*bi;
      x[i]:= c1[k]*y0;
      c0:= c1;
   END;
END;

PROCEDURE lss_triangular( VAR A : rmatrix; VAR b,x : ivector );

{ interval forward substitution for lower triangular system Ax=b  or        }
{ interval backward substitution for upper triangular system Ax=b .         }
{ A must be [1..n,-l..0] or [1..n,0..k]                                     }

VAR i,j,n,k,l : integer;
BEGIN
   n:=      ub(A,1);
   l:= abs( lb(A,2) );
   k:=      ub(A,2);
   IF k = 0 THEN  lss_lower( A, b,x, l )  { A is  lower  triangular matrix }
   ELSE
   IF l = 0 THEN  lss_upper( A, b,x, k )  { A is  upper  triangular matrix }
   ELSE
      { Error:  A is not a regular triangular matrix !   } ;
END;

PROCEDURE lu_decomp( VAR A,Lo,Up : rmatrix; VAR LU_A : imatrix );
{
  compute approximate LU-decomposition of A without pivoting,
  store factors in Lo,Up and store enclosure of Defekt LU - A in LU_A
  A and LU_A  must be [1..n,-l..k]
```

```
  Lo           must be [1..n,-l..0]
  Up           must be [1..n, 0..k]
}
VAR i,j,m,n,k,l : integer;
    dot       : dotprecision;
BEGIN
   n:=      ub(A,1);
   l:= abs( lb(A,2) );
   k:=      ub(A,2);
   FOR i:= 1 TO n DO
   BEGIN
      Lo[i,0]:= 1.0;
      FOR j:= i TO min(n,i+max(k,l)) DO
      BEGIN
        IF j-i <= k THEN
        BEGIN
          dot:= #( A[i,j-i] - FOR m:= max(1,i-l,j-k) TO min(i-1,j)
                               SUM ( Lo[i,m-i]*Up[m,j-m] )       );
          Up[i,j-i]  := #*( dot );
          LU_A[i,j-i]:= ##( Up[i,j-i] - dot );
        END;
        IF ( Up[i,0] <> 0.0 ) AND ( i<>j ) AND ( j-i <= l ) THEN
        BEGIN
           dot:= #( A[j,i-j] - FOR m:= max(1,j-l,i-k) TO min(j,i-1)
                                SUM ( Lo[j,m-j]*Up[m,i-m] )       );
           Lo[j,i-j]  := #*( dot ) / Up[i,0];
           LU_A[j,i-j]:= ##( Lo[j,i-j]*Up[i,0] - dot );
        END;
      END;
   END;
END;

PROCEDURE lss( VAR A : rmatrix; b : ivector; VAR x : ivector );
CONST eps = 0.1;
VAR Lo          : rmatrix[lb(A,1)..ub(A,1),lb(A,2)..0      ];
    Up          : rmatrix[lb(A,1)..ub(A,1),      0..ub(A,2)];
    LU_A        : imatrix[lb(A,1)..ub(A,1),lb(A,2)..ub(A,2)];
    b_app,x_app : rvector[lb(A,1)..ub(A,1)];
    defect,z,za : ivector[lb(A,1)..ub(A,1)];
    i,j,k,l,n,m : integer;
BEGIN
   n:=      ub(A,1);
   l:= abs( lb(A,2) );
   k:=      ub(A,2);
   IF ( l = 0 ) OR ( k = 0 ) THEN
   BEGIN
      lss_triangular( A, mid(b), x_app );
      FOR i:= 1 TO n DO
         defect[i]:= ##( b[i] - FOR j:= max(1,i-l) TO min(n,i+k)
                                 SUM ( A[i,j-i]*x_app[j] )       );
      lss_triangular( A, defect, z );
      x:= x_app + z;
   END ELSE
   BEGIN
      { Compute LU-factorization and Defect LU-A : }
      lu_decomp( A,Lo,Up,LU_A );

      { Compute approximate solution x_app: }
      lss_triangular( Lo,mid(b),b_app );
      lss_triangular( Up,b_app,x_app );

      { compute defect := b - A*x_app of approx. solution x_app : }
      FOR i:= 1 TO n DO
         defect[i]:= ##( b[i] - FOR j:= max(1,i-l) TO min(n,i+k)
```

```
                                  SUM ( A[i,j-i]*x_app[j] )       );
      z:= defect;
      m := 0;                     { Iteration until inclusion is obtained }
      REPEAT                      { or max. iteration count exceeded :    }
         za:= blow( z, eps );
         FOR i:= 1 TO n DO
            z[i]:= ##( defect[i] + FOR j:= max(1,i-l) TO min(n,i+k)
                                     SUM ( LU_A[i,j-i]*za[j] )    );
         lss_triangular( Lo,z,x );
         lss_triangular( Up,x,z );
         m:= m + 1;
      UNTIL ( z IN za ) OR ( m = 10 );


      x:= x_app + z;
   END;
END;

PROCEDURE read_Ab( VAR A : rmatrix; VAR b : ivector );
{ read matrix A and right hand side b }
VAR i,j,k,l,m : integer;
    x        : rvector[1..n];
    c        : char;
    xk       : real;
BEGIN
   { read A : }
   writeln( ' A = ' );
   l:= abs(lb(a,2));
   k:=     ub(a,2) ;
   FOR j:= -l TO k DO
   BEGIN
      read( a[max(1,1-j),j] );
      FOR i:= 2 TO n-abs(j) DO  a[max(i,i-j),j]:= a[max(1,1-j),j];
   END;
   { allow for changes of individual elements of A : }
   write('change elements ? (y/n) ');
   readln; read(c);
   IF c IN ['j','J','y','Y'] THEN
   REPEAT
      write('row, col, new value : '); read(i,j);
      IF (i>0) AND (j>0) THEN read(a[i,j-i]);
   UNTIL ( i=0 ) OR ( j=0 );


   { read b : }
   writeln( ' b = ');

   WHILE eoln OR (input↑=' ') DO  get(input);

   IF input↑ = '-' THEN
   BEGIN
      { determine b such that solution is approx. x[i] = (-1)↑(i+1)*1/i }
      get(input);
      writeln('solution will be (-1)↑(i+1)*1/i');
      FOR i:= 1 TO n DO IF odd(i) THEN x[i]:=  1/i
                                  ELSE x[i]:= -1/i;
      FOR i:= 1 TO n DO
         b[i]:= #*( FOR j:= max(1,i-l) TO min(n,i+k) SUM ( A[i,j-i]*x[j] ) );
   END ELSE
   IF input↑ = 'e' THEN
   BEGIN
      {determine b such that solution is approx. x[i]=10↑(m*(n+1-2*i)/(n-1))}
      get(input);
      writeln('solution will be x[i] = 10↑( m*(n+1-2*i)/(n-1) )');
```

```
      write  ('enter exponent range m = '); read(m);
      xk:= power(10,m/(n-1));
      FOR i:= 1 TO n DO x[i]:= power(xk,n+1-2*i);
      FOR i:= 1 TO n DO
         b[i]:= #*( FOR j:= max(1,i-l) TO min(n,i+k) SUM ( A[i,j-i]*x[j] ) );
   END ELSE
   IF input↑ = '=' THEN
   BEGIN
      { set all components of b equal to b[1] }
      get(input);
      read( b[1] );
      FOR i:= 2 TO n DO b[i]:= b[1];
   END
   ELSE
      { read each b[i] separately }
      FOR i:= 1 TO n DO  read( b[i] );

   { allow for changes of individual elements of b : }
   write('change elements ? (y/n) ');
   readln; read(c);
   IF c IN ['j','J','y','Y'] THEN
   REPEAT
      write('row, new value : '); read(i);
      IF i>0 THEN read(b[i]);
   UNTIL i=0;

END;

PROCEDURE write( VAR f : text; x : ivector );
VAR i : integer;
BEGIN
   writeln(f);
   FOR i:= lb(x) TO ub(x) DO writeln( f, i:3, ': ', x[i] );
   writeln(f);
END;

PROCEDURE main;
VAR A    : rmatrix[1..n,-l..k];
    b,x  : ivector[1..n];
    xr   : rvector[1..n];
    nrm,e_rel  : real;
    i,i_rel,i_abs,i_min,i_max,total : integer;
BEGIN
   read_Ab( A, b );

   lss( A, b, x );

   writeln('x = ');
   IF n<=20 THEN writeln(x)
            ELSE writeln(x[1..10],x[n-10..n]);
   i_min:= 1;
   i_max:= 1;
   e_rel:= 0.0;
   i_abs:= 1;
   i_rel:= 0;
   FOR i:= 1 TO n DO
   BEGIN
      IF sup(abs(x[i])) < sup(abs(x[i_min])) THEN i_min:= i;
      IF inf(abs(x[i])) > inf(abs(x[i_max])) THEN i_max:= i;
      IF NOT ( 0.0 IN x[i] ) THEN BEGIN
                                  nrm:= diam(x[i])/inf(abs(x[i]));
                                  IF nrm > e_rel THEN BEGIN
                                                      e_rel:= nrm;
                                                      i_rel:= i;
```

```
                                                END;
                              END;
      IF diam(x[i]) > diam(x[i_abs]) THEN i_abs:= i;
   END;
   writeln;
   writeln('max. rel. error = ', e_rel, '  at i = ', i_rel );
   writeln('max. abs. error = ', diam(x[i_abs]), '  at i = ', i_abs );
   writeln('min. abs. x[',i_min,'] = ', x[i_min] );
   writeln('max. abs. x[',i_max,'] = ', x[i_max] );
   writeln;
END;

BEGIN
   REPEAT
      write( ' Dimension  n = ' ); read(n);
      IF n>0 THEN
      BEGIN
         writeln( ' Bandwidths l,k :  ' );
         read( l,k );
         main;
      END;
   UNTIL n<=0;
END.
```

2.2.3.3 Test Results

We now present some results which were computed with the program from the previous Subsection.

First we turn back to Example 3 from Section 2.2.1. From interval forward substitution we had there for $x_0 = 0.99$ the enclosure $T_{50}(0.99) \in [-710, 720]$. If we solve the same system (of dimension $n = 51$) with our program we get the following output:

```
 Dimension  n = 51
 Bandwidths l,k :
2 0
 A =
1 -1.98 1
change elements ? (y/n) y
row, col, new value : 2 1 0
row, col, new value : 0 0
 b =
=0
change elements ? (y/n) y
row, new value : 1 1
row, new value : 2 0.99
row, new value : 0 0
x =

  1: [  1.000000000000000E+000,  1.000000000000000E+000 ]
  2: [  9.899999999999999E-001,  9.900000000000002E-001 ]
  3: [  9.601999999999999E-001,  9.602000000000003E-001 ]
  4: [  9.111959999999998E-001,  9.111960000000004E-001 ]
  5: [  8.439680799999997E-001,  8.439680800000004E-001 ]
  6: [  7.598607983999997E-001,  7.598607984000004E-001 ]
  7: [  6.605563008319996E-001,  6.605563008320004E-001 ]
  8: [  5.480406772473596E-001,  5.480406772473604E-001 ]
  9: [   4.24564240117772E-001,   4.24564240117774E-001 ]
 10: [   2.92596518185829E-001,   2.92596518185831E-001 ]
```

```
 41: [  8.129440470900618E-001,  8.129440470900625E-001 ]
 42: [  8.869640273620158E-001,  8.869640273620163E-001 ]
 43: [  9.432447270867294E-001,  9.432447270867298E-001 ]
 44: [  9.806605322697085E-001,  9.806605322697088E-001 ]
 45: [  9.984631268072933E-001,  9.984631268072935E-001 ]
 46: [  9.962964588087321E-001,  9.962964588087323E-001 ]
 47: [  9.742038616339963E-001,  9.742038616339967E-001 ]
 48: [  9.326271872265805E-001,  9.326271872265809E-001 ]
 49: [  8.723979690746331E-001,  8.723979690746337E-001 ]
 50: [  7.947207915411930E-001,  7.947207915411937E-001 ]
 51: [  7.011491981769291E-001,  7.011491981769298E-001 ]


max. rel. error =  5.694110624237756E-014  at i = 12
max. abs. error =  8.326672684688674E-016  at i = 32
min. abs. x[12] = [    1.3861676276707E-002,     1.3861676276709E-002 ]
max. abs. x[1] = [  1.000000000000000E+000,  1.000000000000000E+000 ]
```

Thus we get $T_{50}(0.99) \in 0.7011491981769 29_1^8$ to almost full machine accuracy. Also for higer dimensions we still get excellent enclosures, e.g. $T_{200}(0.99) \in -0.9994368897931 87_4^1$, $T_{1000}(0.99) \in -0.9859539291500 54_5^0$ or $T_{5000}(0.99) \in -0.6681925795233 40_9^1$.

As a second example we compute an enclosure for a very large system. We take the symmetric Toeplitz matrix with five bands having the values $1, 2, 4, 2, 1$ and on the right hand side we set all components of b equal to 1. Then the program produces the following output for a system of size $n = 200000$ (only the first ten and last ten solution components are printed):

```
 Dimension  n = 200000
 Bandwidths l,k :
2 2
 A =
1 2 4 2 1
n
=1
n
change elements ? (y/n)  b =
change elements ? (y/n) x =

  1: [  1.860146067479180E-001,  1.860146067479181E-001 ]
  2: [  9.037859550210300E-002,  9.037859550210302E-002 ]
  3: [  7.518438200412189E-002,  7.518438200412191E-002 ]
  4: [  1.160876404875081E-001,  1.160876404875082E-001 ]
  5: [  1.003153932563721E-001,  1.003153932563722E-001 ]
  6: [  9.427129202687645E-002,  9.427129202687647E-002 ]
  7: [  1.028361799416204E-001,  1.028361799416205E-001 ]
  8: [  1.005240450090008E-001,  1.005240450090009E-001 ]
  9: [  9.874921290539136E-002,  9.874921290539138E-002 ]
 10: [  1.004617422430963E-001,  1.004617422430964E-001 ]


199990: [  1.001953939326196E-001,  1.001953939326197E-001 ]
199991: [  1.004617422430963E-001,  1.004617422430964E-001 ]
199992: [  9.874921290539136E-002,  9.874921290539138E-002 ]
199993: [  1.005240450090008E-001,  1.005240450090009E-001 ]
199994: [  1.028361799416204E-001,  1.028361799416205E-001 ]
199995: [  9.427129202687645E-002,  9.427129202687647E-002 ]
199996: [  1.003153932563721E-001,  1.003153932563722E-001 ]
```

```
199997: [  1.160876404875081E-001,  1.160876404875082E-001 ]
199998: [  7.518438200412189E-002,  7.518438200412191E-002 ]
199999: [  9.037859550210300E-002,  9.037859550210302E-002 ]
200000: [  1.860146067479180E-001,  1.860146067479181E-001 ]


max. rel. error =  1.845833860422451E-016  at i = 3
max. abs. error =  2.775557561562891E-017  at i = 1
min. abs. x[3] = [  7.518438200412189E-002,  7.518438200412191E-002 ]
max. abs. x[1] = [  1.860146067479180E-001,  1.860146067479181E-001 ]
```

Chapter 3

Eigenvalue Problems

In this chapter we will demonstrate methods for the inclusion of eigenvalues and eigenvectors of square matrices. We will use two different approaches:

1. A general one which can be applied to any eigenvalue problem which is treated as a nonlinear system of equations by this approach. Here we need, however, a starting approximation for both, an eigenvalue and the corresponding eigenvector. The method then encloses an eigenpair which lies close to the given starting approximation.

2. A method more specialized to symmetric eigenvalue problems which encloses all eigenvalues and eigenvectors of a symmetric matrix. Here we do not need starting approximations. The method will enclose all eigenvalues and eigenvectors of the given symmetric matrix.

Both methods can be equally applied to real, interval, complex and complex interval matrices.

The first method is due to Rump, [280], [285]. We will apply it to the ordinary eigenvalue problem

$$Ax = \lambda x \tag{3.1}$$

as well as to the general eigenvalue problemi

$$Ax = \lambda Bx \quad . \tag{3.2}$$

This method can be viewed as a Newton like interval iteration method for the nonlinear system

$$\begin{pmatrix} Ax - \lambda x \\ e_k^T x - 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

in the case of (3.1) and for the system

$$\begin{pmatrix} Ax - \lambda Bx \\ e_k^T x - 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

in the case of (3.2). (Here e_k^T is the k-th unit vector and the last equation in these systems means that the k-th component of the eigenvector x is normalized to be 1). Therefore, as mentioned already, we need starting approximations for both the eigenvalues and the eigenvectors. Then only one eigenpair will be enclosed. If we want to enclose all eigenvalues and eigenvectors of the matrix we also must supply approximations for all of them. A drawback of the method is, that it works only for simple eigenvalues. I.e. multiple eigenvalues and their eigenvectors cannot be enclosed by this method. However, to the authors knowledge for the ordinary eigenvalue problem there does not yet exist a general method which can enclose multiple eigenvalues and their eigenvectors without prior knowledge of their multiplicity.

The second method can be applied to symmetric matrices only, it can also be generalized to complex Hermitian matrices. It consists of two basic steps: First an approximate eigensystem is determined (by use of the Jacobi method in our case). Second, a similarity transformation of A is performed with this approximation and the resulting, usually strongly diagonally dominant matrix is taken to enclose the eigenvalues by use of the Gerschgorin circle theorem. In this second step all roundoff errors are taken into account. Finally, for the enclosure of the eigenvectors we can use standard estimations from Wilkinson, [333], or others. In the first step we can refine the approximate eigensystem by some sort of residual iteration which was introduced in Lohner, [231].

The differences of this second method from the previous one are that it encloses all eigenvalues and eigenvectors of the symmetric matrix simultaneously and independently of the multiplicities of the eigenvalues (which are all real here) and that this method does not need any approximations of the eigenvalues and eigenvectors.

The first method will be applied to the ordinary eigenvalue problem in the next Section 3.1 and to the general eigenvalue problem in Section 3.2. Then, in Section 3.3 we will apply the second method to the symmetric eigenvalue problem.

Other approaches to the verified solution of eigenvalue problems can be found in the work of Goerisch and Behnke, [34], [35], [36], [112]. The methods developed there are based on variational principles and more general theorems and are therefore also suited for the application to eigenvalue problems with differential and integral equations.

3.1 The Eigenvalue Problem $Ax = \lambda x$

3.1.1 Introduction

Here we discuss a method for the computation of enclosures of a simple eigenvalue λ and its associated eigenvector x of the ordinary matrix eigenvalue problem (3.1). This enclosure method was introduced by Rump, [280], [285], and can be interpreted as a Newton like iteration method for a zero of the function $f : I\!R^{(n+1)} \to I\!R^{(n+1)}$ with

$$f(x, \lambda) := \begin{pmatrix} Ax - \lambda x \\ e_k^T x - 1 \end{pmatrix} \tag{3.3}$$

and an $n \times n$-matrix A. See also [244] for a discussion of this method.

The last component of f forces the eigenvector x to have the k-th component equal to one (e_k is the k-th unit-vector). Any other normalization of x could also be used (e.g. $x^T x - 1 = 0$), however the one we have chosen makes some things easier to be written down and simplifies the computation somewhat.

We stress again that for this method we need an approximate eigenvalue $\tilde{\lambda}$ and an approximate eigenvector $\tilde{x}$ and that this method can only be used to enclose eigenpairs corresponding to a simple eigenvalue λ.

The approximations which are needed can be computed by any method, a popular one which can also be found in many software libraries is the QR-method, see e.g. [309] or [308].

The method can be easily extended to different data types like interval, complex and complex interval.

3.1.2 Theoretical Background

Assume we have an approximation $\tilde{\lambda}$ of an eigenvalue λ and $\tilde{x}$ of the corresponding eigenvector x of the eigenvalue problem (3.1). Then we denote the error of these approximations by

$$\begin{pmatrix} y \\ \mu \end{pmatrix} = \begin{pmatrix} x - \tilde{x} \\ \lambda - \tilde{\lambda} \end{pmatrix} . \tag{3.4}$$

Substituting $x = \tilde{x} + y$ and $\lambda = \tilde{\lambda} + \mu$ in (3.3) we get

$$\begin{aligned} f(x, \lambda) &= f(\tilde{x} + y, \tilde{\lambda} + \mu) \\ &= \begin{pmatrix} A(\tilde{x} + y) - (\tilde{\lambda} + \mu)(\tilde{x} + y) \\ e_k^T(\tilde{x} + y) - 1 \end{pmatrix} \\ &= \begin{pmatrix} A\tilde{x} - \tilde{\lambda}\tilde{x} \\ e_k^T \tilde{x} - 1 \end{pmatrix} + \begin{pmatrix} Ay - \tilde{\lambda} y - \mu(\tilde{x} + y) \\ e_k^T y \end{pmatrix} \\ &= f(\tilde{x}, \tilde{\lambda}) + \begin{pmatrix} A - \tilde{\lambda} I & -(\tilde{x} + y) \\ e_k^T & 0 \end{pmatrix} \begin{pmatrix} y \\ \mu \end{pmatrix} \end{aligned}$$

where I is the $n \times n$ identity matrix.

Multiplying $f(x, \lambda) = 0$ with this representation of f by an $(n+1) \times (n+1)$-matrix $-C$ and adding $(y, \mu)^T$ to both sides we obtain an equivalent equation which is in fixed point form.

With the function

$$g(y, \mu) = -Cf(\tilde{x}, \tilde{\lambda}) + (I_{n+1} - C \begin{pmatrix} A - \tilde{\lambda} I & -(\tilde{x} + y) \\ e_k^T & 0 \end{pmatrix}) \begin{pmatrix} y \\ \mu \end{pmatrix} \tag{3.5}$$

where I_{n+1} is the $(n+1) \times (n+1)$ identity matrix, this fixed point form is

$$\begin{pmatrix} y \\ \mu \end{pmatrix} = g(y, \mu) . \tag{3.6}$$

If C is non-singular then the fixed points (y^*, μ^*) of g are just the values for which $x = \tilde{x} + y^*$ and $\lambda = \tilde{\lambda} + \mu^*$ are the eigenpairs of A. We will compute enclosures for these fixed points by use of an interval iteration based on (3.6):

$$\begin{pmatrix} [y_{k+1}] \\ [\mu_{k+1}] \end{pmatrix} = g([y_k], [\mu_k]), \quad k \geq 0 \;, \tag{3.7}$$

with a suitable starting interval $([y_0], [\mu_0])^T$, see below. Then in (3.5) we must of course use interval operations to evaluate g.

If for this interval iteration we have an index k for which an interval vector $([y_k], [\mu_k])^T$ is mapped into its interior

$$\begin{pmatrix} [y_{k+1}] \\ [\mu_{k+1}] \end{pmatrix} = g([y_k], [\mu_k]) \subseteq \quad \mathrm{int} \begin{pmatrix} [y_k] \\ [\mu_k] \end{pmatrix} \tag{3.8}$$

then we can apply Brouwer's fixed point theorem to prove the existence of a fixed point of g.

If condition (3.8) holds then we can even show the following assertions, which are proved e.g. in [280], [285] or in [244] :

1. The matrix C is non-singular.
2. In $(\tilde{x} + [y_k], \tilde{\lambda} + [\mu_k])$ there is a unique eigenpair of A.
3. The corresponding eigenvalue is simple.
4. If iteration (3.7) is continued with the interval $([y_k], [\mu_k])^T$ from (3.8) then the iterates are contained successively in each other and they converge monotonely decreasing to the enclosed unique eigenpair.

For the practical application of this iteration on the computer we must, however, specify some steps more detailed: How do we choose the matrix C? Which starting vector $([y_0], [\mu_0])^T$ do we use? Can we guarantee that at some index we achieve condition (3.8)?

A good choice for C is usually an approximate inverse of the matrix

$$R := \begin{pmatrix} A - \tilde{\lambda} I & -\tilde{x} \\ e_k^T & 0 \end{pmatrix} \tag{3.9}$$

i.e. $C \approx R^{-1}$ since then the iteration matrix $I_{n+1} - C(R - \left(\begin{smallmatrix} 0 & y \\ 0 & 0 \end{smallmatrix}\right))$ in (3.5), (3.6) will be close to the zero matrix if $\tilde{\lambda}$ and $\tilde{x}$ are sufficiently good (i.e. y is sufficiently small).

The starting vector for the iteration can be simply chosen to be the zero vector, $([y_0], [\mu_0])^T = (0, 0)^T$. In the next iteration then $([y_1], [\mu_1])^T = -Cf(\tilde{x}, \tilde{\lambda})$, so this vector is a similarly simple starting vector which we will use in our algorithm.

Finally, in order to achieve condition (3.8) at some iteration index k we introduce an ϵ-inflation in the same way as we have used it already in (2.8) in Section 2.1 of Chapter 2. Thus in each iteration step before applying the iteration function g we first inflate the old iterate by a certain amount.

In order to improve the accuracy of the results and to keep the number of interval iterations small we introduce additionally a floating-point iteration to improve the quality of the approximations $\tilde{x}$ and $\tilde{\lambda}$

$$\begin{pmatrix} \tilde{x}_{k+1} \\ \tilde{\lambda}_{k+1} \end{pmatrix} = \begin{pmatrix} \tilde{x}_{k} \\ \tilde{\lambda}_{k} \end{pmatrix} - Cf(\tilde{x}_k, \tilde{\lambda}_k) \tag{3.10}$$

until some stopping criterion is satisfied. After this iteration we redefine $\tilde{x}$ and $\tilde{\lambda}$ to be the final values from this floating-point iteration.

Altogether we now obtain the following steps for our enclosure method.

1. Compute an approximate eigenpair $(\tilde{x}, \tilde{\lambda})$ (using any available software).
2. Compute an approximate inverse $C \approx R^{-1}$, with R from (3.9).
3. Improve $\tilde{x}$ and $\tilde{\lambda}$ by performing the floating-point iteration (3.10).
4. Set $([y_0], [\mu_0])^T = (0, 0)^T$.
5. Iterate $([\hat{y}_k], [\hat{\mu}_k])^T = blow([y_k], [\mu_k])^T$, $([y_{k+1}], [\mu_{k+1}])^T = g([\hat{y}_k], [\hat{\mu}_k])$ until an index k is reached for which $([y_{k+1}], [\mu_{k+1}])^T \subseteq \ \mathrm{int}([\hat{y}_k], [\hat{\mu}_k])^T$, i.e. condition (3.8) is satisfied, or until some specified maximal iteration count is exceeded.
6. If (3.8) was satisfied then $(\tilde{x} + [y_k], \tilde{\lambda} + [\mu_k])^T$ is the desired enclosure of a simple eigenpair of A. Otherwise the algorithm did not succeed (e.g. the approximation of the eigenpair was too bad, or the eigenvalue is not simple of the problem is too bad conditioned).

We note that in the interval iteration we do not have to recompute the iteration matrix in each step completely. Since only the last column of this matrix depends on the iterates, see (3.5), we can compute columns $1, \ldots, n$, prior to the iteration. Then in each step we only compute the $(n+1)$-st column $e_{n+1}^T - C\binom{-(\tilde{x}+y)}{0}$.

Remark:

As is the case of dense systems of linear equations we can use for C an approximate inverse of double length $C = C_1 + C_2$, compare Section 2.1, which can be computed in the same way as in (2.10). With such a double length approximate inverse many more ill-conditioned eigenpairs can be computed as with a single length C only.

3.1.3 Algorithms

The algorithm for the computation of enclosures for the eigenvalue and eigenvector of a real matrix A according to the previous Section is listed below. The floating-point approximation $\tilde{x}$ of the eigenvector is assumed to be normalized to one in the k-th component already.

Algorithm 3.1: Eig $(A, lam, rx, ilam, ix, errcode)$ {procedure}

{Enclosure of an eigenpair for the eigenvalue problem $Ax = \lambda x$}
input: real matrix A, real lam (appr. eigenvalue), real vector rx (appr. eigenvector)
output: interval $ilam$ (eigenvalue enclosure), interval vector ix (eigenvector enclosure), integer $errcode$ (error indicator)

(a) initialize errcode :
$errcode := 0$

(b) compute approximate inverse :
$R := \begin{pmatrix} A - lam \cdot I & -rx \\ e_k^T & 0 \end{pmatrix}$
$C :=$Minv$(R, errcode)$
if $errcode \neq 0$ **then** stop

(c) improve floating-point approximation :
$(rx_{old}, lam_{old}) := (rx, lam)$
repeat
$(rx, lam) := (rx, lam) - C \cdot (A \cdot rx_{old} - lam_{old} \cdot rx_{old}, e_k \cdot rx_{old} - 1)$
until (rx, lam) accurate enough **or** max. iteration count exceeded

(d) compute enclosure of defect of floating-point approximation :
$D := -C \cdot \begin{pmatrix} \Diamond(A \cdot rx - lam \cdot rx) \\ 0 \end{pmatrix}$

(e) perform interval iteration :
$Z := D$ { starting vector }
$M := \mathrm{id}(C) - C \cdot \begin{pmatrix} \Diamond(A - lam \cdot I) & -rx \\ e_k^T & 0 \end{pmatrix}$ { iteration matrix }
repeat
{ compute last column of iteration matrix : }
$M_{*,n+1} := e_{n+1}^T - C \cdot \begin{pmatrix} -rx - Z_{1..n} \\ 0 \end{pmatrix}$
$Z_A := \mathrm{blow}(Z, \epsilon)$ { ϵ-inflation }
$Z := D + M \cdot Z_A$
until $Z \subseteq \mathrm{int}(Z_A)$ **or** max. iteration count exceeded

(f) return result :
if $Z \subseteq \mathrm{int}(Z_A)$ **then**
$ilam := lam + Z_{n+1}$ { eigenvalue enclosure }
$ix := rx + Z_{1..n}$ { eigenvector enclosure }
else
$errcode := 1$

(g) **return** $ilam, ix, errcode$

3.1.4 PASCAL-XSC Program Code

The PASCAL-XSC program code of the method is listed in the following module `eig` for the data type real. The changes which have to be made in order to get modules `ieig`, `ceig` and `cieig` for the solution of eigenvalue problems with interval, complex and complex interval data types, resp., are almost the same as the changes which were necessary to convert the module `lss` in Section 2.1 to these data types. To do this conversion the changes mentioned there should be applied appropriately to the module `eig`.

The program in this module differs from the previous algorithm slightly in its data structure: instead of using $(n+1)$-vectors and $(n+1) \times (n+1)$-matrices we replace the k-th component of x by the eigenvalue λ since this component does never change because of the normalization of the eigenvector. Similarly the k-th column of the iteration matrix is replaced by the $(n+1)$-st column of the previous notation. The $(n+1)$-st row of this matrix can be omitted completely since it is never used. Thus all matrices and vectors in the program are of dimension n only.

```
MODULE eig;

USE  i_ari,mvi_ari,mv_ari,lss_aprx;

GLOBAL
PROCEDURE EIG( VAR A : RMATRIX; LAM : REAL;      VAR RX : RVECTOR;
                        VAR ILAM : INTERVAL; VAR IX : IVECTOR;
               VAR errcode : INTEGER );

CONST zerotest = 1E6;             { accuracy constant in REL }
      delta    = 1E-15;           { accuracy constant for the rel. error }
      sqrt_01  = 0.31622777;
      abort    = 1E20;            { accuracy constant in TOO_BAB }

VAR I,K,KO,N,ERR   : INTEGER;
    C              : RMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    RD,X,XO        : RVECTOR[LB(A)..UB(A)];
    FU,MB          : IMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    D,Z,ZA,VIID_KO : IVECTOR[LB(A)..UB(A)];
    P,bound,eps    : REAL;
    ZA_KO          : INTERVAL;
    READY          : BOOLEAN;

FUNCTION REL ( VAR A,B : RVECTOR ) : REAL;
{-----------------------------------------------------------------------}
{ computes componentwise the maximum relative error of A  w.r.t  B.     }
{ If A[i] and B[i] do not have the same sign or if B[i] = 0, then       }
{ rel. error := 0 for this component.                                   }
{ A is always the new value of an iteration, B the old one.             }
{-----------------------------------------------------------------------}
VAR i         : INTEGER;
    p,r,ai,bi : REAL;
BEGIN
   p:= 0.0;
   FOR i:= LB(A) TO UB(A) DO      { A,B must have same index range }
   BEGIN
      ai:= A[i];
      bi:= B[i];
      IF ( ai*bi <= 0.0 )  OR  ( zerotest*ABS(ai) < ABS(bi) )
                     THEN   r:= 0.0
                     ELSE   r:= ABS( (ai-bi)/bi );
      IF r>p THEN p:= r;
   END;
   REL:= p;
END; { REL }
```

```
FUNCTION TOO_BAD ( VAR A : IVECTOR ) : BOOLEAN;
{ TOO_BAD := accuracy of A is far too bad }
VAR i   : INTEGER;
    bad : BOOLEAN;
BEGIN
   bad:= FALSE;
   FOR i:= LB(A) TO UB(A) DO
      bad:= bad OR (A[i].INF<-abort) AND (A[i].SUP>abort);
   TOO_BAD := bad;
END; { TOO_BAD }


BEGIN
   errcode:= 0;
   N:= UB(A,1) - LB(A,1) + 1;
   {-------------------------------------------------}
   {   errcode:= 3 : Matrix A not square             }
   {   errcode:= 4 : Vector RX has wrong dimension   }
   {   errcode:= 5 : Vector IX has wrong dimension   }
   {-------------------------------------------------}
   IF n <> ( UB(A,2) - LB(A,2) + 1 ) THEN errcode:= 3;
   IF n <>  ( UB(RX) - LB(RX) + 1 )  THEN errcode:= 4;
   IF n <>  ( UB(IX) - LB(IX) + 1 )  THEN errcode:= 5;

   IF errcode = 0 THEN
   BEGIN
      P:= 0.0;
      X:= RX;
      { find largest component in eigenvector approximation RX : }
      { keep index of this component in K0                       }
      FOR I:= LB(A) TO UB(A) DO IF ABS(X[I])>P THEN BEGIN
                                                 P:= ABS(X[I]);
                                                 K0:= I;
                                              END;

      { K0-th unit vector : }
      VIID_K0    := 0.0;
      VIID_K0[K0]:= 1.0;

      { rescale approx. eigenvector such that largest component is = 1.0 }
      X:= (1.0/X[K0]) * X;

      READY:= TRUE;

      { compute approximate inverse : }
      X[K0]   :=  1.0;
      C       :=  #*( A - LAM*ID(A) );
      C[*,K0] :=  -X;
      MINV( C,ERR );
      READY:= ERR = 0;

      IF NOT ready THEN  errcode:= 2   { matrix probably singular }
      ELSE
      BEGIN
         { use K0-th component for computation of the eigenvalue }
         X[K0]   :=  LAM;

         { improve floating point approximation: }
         K:= 0;  bound:= 100.0*sqrt_01;
         REPEAT
            K:= K+1;

            { compute defect : }
```

```
          X[K0]:= 1.0;
          RD   := #*( A*X - LAM*X );
          X[K0]:= LAM;

          X0:= X;
          X := #*( X - C*RD );

          P  := REL( X,X0 );
          LAM:= X[K0];
          bound:= bound * sqrt_01;
       UNTIL (K>=2) AND ((K>=20) OR (P>=bound) OR (P<delta));

       { compute defect of improved approximation : }
       X[K0]:= 1.0;
       RD:=   #*( A*X - LAM*X );        { Decomposition of the term      }
       D :=   ##( A*X - LAM*X - RD );   { A*X-LAM*X into the sum RD + D }
       D := - ##( C*RD + C*D );
       Z :=   D;
       X[K0]:= LAM;

       { residual :  E  -  R * funktional matrix }
       FU := ##( A - LAM*ID(A) );
       MB := ##( ID(A) - C*FU );

       { interval iteration : }
       K    := 0;
       eps  := 0.05;
       READY:= FALSE;
       X[K0]:= 1.0;
       REPEAT
          K:= K+1;
          IF K>=5 THEN eps := 10.0*eps;

          ZA:= BLOW( Z, eps );

          { compute K0-th column of functional matrix : }
          ZA_K0    :=  ZA[K0];
          ZA[K0]   :=  0.0;
          FU[*,K0]:= -( X + ZA );
          ZA[K0]   :=  ZA_K0;
          MB[*,K0]:=  ##( VIID_K0 - C*IVECTOR(FU[*,K0]) );

          Z:=  D + MB*ZA;
          READY:= Z IN ZA;

       UNTIL READY OR (K>=10) OR TOO_BAD(Z);

       ILAM := LAM + Z[K0];
       Z[K0]:= 0.0;

       IX   := X + ORD(READY) * Z;
       errcode:= ORD( NOT READY );

    END { if not ready then ... else};

  END { errcode<>0 };

END (* EIG *);

END.
```

3.1.5 Test Results

We demonstrate the use of the module `eig` by the following short PASCAL-XSC program which reads in a matrix A and eigenvalue and eigenvector approximations. Procedure `EIG` from the module `eig` is then used to enclose an eigenpair which is close to the given approximation. The approximations can be entered repeatedly until '`n`' or '`N`' is entered in procedure `main`. Then a new dimension n and another matrix A may be entered, otherwise the program stops (if $n \leq 0$ is entered).

```
PROGRAM eigtest;

USE i_ari,mv_ari,mvi_ari,eig;

PROCEDURE MAIN( N : INTEGER );

VAR ERR : INTEGER;
    A   : RMATRIX[1..N,1..N];
    EV  : RVECTOR[1..N];
    EW  : REAL;
    IEV : IVECTOR[1..N];
    IEW : INTERVAL;
    OK  : BOOLEAN;
    C   : CHAR;

PROCEDURE WRITE_EW;
VAR J : INTEGER ;
BEGIN
   WRITELN;
   WRITELN('Eigenvalue  = ', IEW );
   WRITELN('Eigenvector = ', IEV );
END { WRITE_EW };

BEGIN
   WRITELN('A = '); READ(A);

   REPEAT
      WRITE('Eigenvalue  approximation : ');  READ(EW);
      WRITE('Eigenvector approximation : ');  READ(EV);

      EIG( A,EW,EV,IEW,IEV,ERR );
      OK:= ERR=0;

      IF OK THEN  WRITE_EW  ELSE
      BEGIN
         WRITELN;
         WRITELN( 'Verification not succeeded. errcode = ', ERR );
      END;

      WRITELN;
      WRITE( 'another eigenvalue ? (y/n) ' );
      READLN; READ(C);

   UNTIL C IN ['N','n'];
END;

VAR N : INTEGER ;

BEGIN
   REPEAT
      WRITE(' N ? (End = 0) '); READ(N);
      IF N>0 THEN MAIN( N );
```

```
    UNTIL N<=0;
 END.
```

As an example we use the 3×3 matrix from [113]

$$\begin{pmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{pmatrix}$$

with the eigenvalues $\lambda_1 = 1, \lambda_2 = 2, \lambda_3 = 3$ and the corresponding eigenvectors $x_1 = (-15, 12, 4)^T, x_2 = (-16, 13, 4)^T, x_3 = (-4, 3, 1)^T$.

Taking the approximations $\tilde{\lambda}_1 = 1.01, \tilde{x}_1 = (-15.01, 11.99, 3.99)^T$, $\tilde{\lambda}_2 = 1.99, \tilde{x}_2 = (-15.99, 13.01, 4.01)^T$ and $\tilde{\lambda}_3 = 3.01, \tilde{x}_3 = (-4.01, 2.99, 0.99)^T$ as input for our test program we obtain the following output:

```
 N ? (End = 0) 3
A =
 33  16  72
-24 -10 -57
 -8  -4 -17
Eigenvalue  approximation : 1.01
Eigenvector approximation : -15.01 11.99 3.99

Eigenvalue  = [  9.999999999999998E-001,  1.000000000000001E+000 ]
Eigenvector =
[  1.000000000000000E+000,  1.000000000000000E+000 ]
[ -8.000000000000001E-001, -7.999999999999999E-001 ]
[ -2.666666666666668E-001, -2.666666666666666E-001 ]


another eigenvalue ? (y/n) y
Eigenvalue  approximation : 1.99
Eigenvector approximation : -15.99 13.01 4.01

Eigenvalue  = [  1.999999999999999E+000,  2.000000000000001E+000 ]
Eigenvector =
[  1.000000000000000E+000,  1.000000000000000E+000 ]
[ -8.125000000000002E-001, -8.124999999999998E-001 ]
[ -2.500000000000001E-001, -2.499999999999999E-001 ]


another eigenvalue ? (y/n) y
Eigenvalue  approximation : 3.01
Eigenvector approximation : -4.01 2.99 0.99

Eigenvalue  = [  2.999999999999999E+000,  3.000000000000001E+000 ]
Eigenvector =
[  1.000000000000000E+000,  1.000000000000000E+000 ]
[ -7.500000000000002E-001, -7.499999999999998E-001 ]
[ -2.500000000000001E-001, -2.499999999999999E-001 ]


another eigenvalue ? (y/n) n
 N ? (End = 0) 0
```

We see that the greatest component of each eigenvector is normalized to one and all enclosures are highly accurate. In fact they are 1 ulp intervals in the internal binary representation and they are only widened somewhat by the directed rounding of the binary to decimal conversion during output.

3.2 The General Eigenvalue Problem $Ax = \lambda Bx$

3.2.1 Introduction

The general eigenvalue problem (3.2) with two matrices $n \times n$-matrices A and B is often considered only for the case of symmetric A and symmetric and positive definite B only. In this case it is guaranteed that there are n eigenvalues (not necessarily distinct) with corresponding eigenvectors.

In the case of arbitrary matrices A and B there may be more or less than n eigenvalues as can be seen from the following simple 2×2-examples.

$$\begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix} x = \lambda \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x$$

has only one simple eigenvalue, $\lambda = 3$ with eigenvector $x = \begin{pmatrix} -2 \\ 1 \end{pmatrix}$.

For

$$\begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} x = \lambda \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} x$$

all $\lambda \in I\!R$ are eigenvalues, the corresponding eigenvectors are $x = \begin{pmatrix} \lambda - 2 \\ 1 \end{pmatrix}$.

The other extreme is

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x = \lambda \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} x$$

which has no eigenvalues at all.

Nevertheless, we will allow A and B to be arbitrary $n \times n$-matrices since the method which we will apply does not need additional hypotheses. It is a slight modification of the method of the previous Section 3.1 and is again only suited for the enclosure of simple eigenvalue/eigenvector pairs.

As for the ordinary eigenvalue problem we seek the solutions as zeros of the quadratic function

$$f(x, \lambda) := \begin{pmatrix} Ax - \lambda Bx \\ e_k^T x - 1 \end{pmatrix} \tag{3.11}$$

3.2.2 Theoretical Background

Since the theory of the method is practically identical to the case of the ordinary eigenvalue problem we will only sketch it briefly. Basically we can copy everything from Section 3.1.2 and replace the identity matrix I in $Ax = \lambda Ix$ by the matrix B.

First we get the following reformulation of (3.11)

$$\begin{aligned}
f(x,\lambda) &= f(\tilde{x}+y,\tilde{\lambda}+\mu)\\
&= \begin{pmatrix} A(\tilde{x}+y)-(\tilde{\lambda}+\mu)B(\tilde{x}+y) \\ e_k^T(\tilde{x}+y)-1 \end{pmatrix}\\
&= \begin{pmatrix} A\tilde{x}-\tilde{\lambda}B\tilde{x} \\ e_k^T\tilde{x}-1 \end{pmatrix} + \begin{pmatrix} Ay-\tilde{\lambda}By-\mu B(\tilde{x}+y) \\ e_k^T y \end{pmatrix}\\
&= f(\tilde{x},\tilde{\lambda}) + \begin{pmatrix} A-\tilde{\lambda}B & -B(\tilde{x}+y) \\ e_k^T & 0 \end{pmatrix} \begin{pmatrix} y \\ \mu \end{pmatrix}
\end{aligned}$$

from which we again obtain a fixed point form

$$\begin{pmatrix} y \\ \mu \end{pmatrix} = g(y,\mu)\ . \tag{3.12}$$

with the function

$$g(y,\mu) = -Cf(\tilde{x},\tilde{\lambda}) + (I_{n+1} - C\begin{pmatrix} A-\tilde{\lambda}B & -B(\tilde{x}+y) \\ e_k^T & 0 \end{pmatrix}) \begin{pmatrix} y \\ \mu \end{pmatrix}\quad . \tag{3.13}$$

Now all details of the method work completely analogous to the case of the ordinary eigenvalue problem with one minor exception: In the function f now, there appears a triple product, λBx, which can no longer be computed directly by use of a scalar product expression.

The only place in the algorithm where we must take care of this triple product is the computation of the defect of the approximate solution,

$$d = A\tilde{x} - \tilde{\lambda}B\tilde{x}\ .$$

For the floating-point approximations $\tilde{x}$ and $\tilde{\lambda}$, however, each component of the product $\tilde{\lambda}\tilde{x}$ has at most a double length mantissa and thus this vector can be split *without roundoff-error* into the sum of two floating-point vectors x_1 and x_2:

$$x_1 + x_2 = \tilde{\lambda}\tilde{x}\ .$$

Now the defect d can be rewritten as follows:

$$d = A\tilde{x} - Bx_1 - Bx_2$$

and this is again a scalar product expression which can be evaluated exactly (or with one rounding only) in PASCAL-XSC.

This modification of the computation of d and the replacement of the identity matrix I by the matrix B at the appropriate places are the only differences between the method for the general eigenvalue problem and the method in Section 3.1 for the ordinary eigenvalue problem.

3.2.3 Algorithms

Obviously the algorithm for the general eigenvalue problem is almost identical to the one in Section 3.1.3. The modified algorithm is as follows.

Algorithm 3.2: Eig $(A, B, lam, rx, ilam, ix, errcode)$ {procedure}

{Enclosure of an eigenpair for the eigenvalue problem $Ax = \lambda Bx$}
input: real matrices A, B, real lam (appr. eigenvalue), real vector rx (appr. eigenvector)
output: interval $ilam$ (eigenvalue enclosure), interval vector ix (eigenvector enclosure), integer $errcode$ (error indicator)

(a) initialize errcode :
$errcode := 0$

(b) compute approximate inverse :
$R := \begin{pmatrix} A-lam\cdot B & -B\cdot rx \\ e_k^T & 0 \end{pmatrix}$
$C := \text{Minv}(R, errcode)$
if $errcode \neq 0$ **then** stop

(c) improve floating-point approximation :
$(rx_{old}, lam_{old}) := (rx, lam)$
repeat
  $(rx, lam) := (rx, lam) - C \cdot (A \cdot rx_{old} - lam_{old} \cdot B \cdot rx_{old}, e_k \cdot rx_{old} - 1)$
until (rx, lam) accurate enough **or** max. iteration count exceeded

(d) compute enclosure of defect of floating-point approximation :
$rx_1 + rx_2 := lam \cdot rx$ { split double-length product in sum of two reals }
$D := -C \cdot \begin{pmatrix} \Diamond(A\cdot rx - B\cdot rx_1 - B\cdot rx_2) \\ 0 \end{pmatrix}$

(e) perform interval iteration :
$Z := D$ { starting vector }
$M := \text{id}(C) - C \cdot \begin{pmatrix} \Diamond(A-lam\cdot B) & -B\cdot rx \\ e_k^T & 0 \end{pmatrix}$ { iteration matrix }
repeat
  { compute last column of iteration matrix : }
  $M_{*,n+1} := e_{n+1}^T - C \cdot \begin{pmatrix} -B\cdot rx - B\cdot Z_{1..n} \\ 0 \end{pmatrix}$
  $Z_A := \text{blow}(Z, \epsilon)$ { ϵ-inflation }
  $Z := D + M \cdot Z_A$
until $Z \subseteq \text{int}(Z_A)$ **or** max. iteration count exceeded

(f) return result :
if $Z \subseteq \text{int}(Z_A)$ **then**
  $ilam := lam + Z_{n+1}$ { eigenvalue enclosure }
  $ix := rx + Z_{1..n}$ { eigenvector enclosure }
else
  $errcode := 1$

(g) **return** $ilam, ix, errcode$

3.2.4 PASCAL-XSC Program Code

As is obvious from the previous discussion the following module **geig** for the computation of enclosures of eigenvalue/eigenvector pairs for the general eigenvalue problem differs only at few places from the corresponding module **eig** in Section 3.1.4. Also the changes which have to be done to make the module work for the data types interval, complex or complex interval are almost literally the same as in Section 3.1.4.

```
MODULE geig;

USE  i_ari,mvi_ari,mv_ari,lss_aprx;

GLOBAL
PROCEDURE EIG( VAR A,B : RMATRIX; LAM : REAL;     VAR RX : RVECTOR;
                            VAR ILAM : INTERVAL; VAR IX : IVECTOR;
               VAR errcode : INTEGER );

CONST zerotest = 1E6;             { accuracy constant in REL }
      delta    = 1E-15;           { accuracy constant for the rel. error }
      sqrt_01  = 0.31622777;
      abort    = 1E20;            { accuracy constant in TOO_BAB }

VAR I,K,K0,N,ERR    : INTEGER;
    C               : RMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    RD,X,X0,RX1,RX2 : RVECTOR[LB(A)..UB(A)];
    FU,MB           : IMATRIX[LB(A)..UB(A),LB(A)..UB(A)];
    D,Z,ZA,VIID_K0  : IVECTOR[LB(A)..UB(A)];
    P,bound,eps     : REAL;
    ZA_K0           : INTERVAL;
    HD              : DOTPRECISION;
    READY           : BOOLEAN;

FUNCTION REL ( VAR A,B : RVECTOR ) : REAL;
{-----------------------------------------------------------------------------}
{ computes componentwise the maximum relative error of A  w.r.t  B.          }
{ If A[i] and B[i] do not have the same sign or if B[i] = 0, then            }
{ rel. error := 0 for this component.                                        }
{ A is always the new value of an iteration, B the old one.                  }
{-----------------------------------------------------------------------------}
VAR i         : INTEGER;
    p,r,ai,bi : REAL;
BEGIN
   p:= 0.0;
   FOR i:= LB(A) TO UB(A) DO     { A,B must have same index range }
   BEGIN
      ai:= A[i];
      bi:= B[i];
      IF ( ai*bi <= 0.0 )  OR  ( zerotest*ABS(ai) < ABS(bi) )
                    THEN   r:= 0.0
                    ELSE   r:= ABS( (ai-bi)/bi );
      IF r>p THEN p:= r;
   END;
   REL:= p;
END; { REL }


FUNCTION TOO_BAD ( VAR A : IVECTOR ) : BOOLEAN;
{ TOO_BAD := accuracy of A is far too bad }
VAR i   : INTEGER;
    bad : BOOLEAN;
```

```
BEGIN
   bad:= FALSE;
   FOR i:= LB(A) TO UB(A) DO
      bad:= bad OR (A[i].INF<-abort) AND (A[i].SUP>abort);
   TOO_BAD := bad;
END; { TOO_BAD }


BEGIN
   errcode:= 0;
   N:= UB(A,1) - LB(A,1) + 1;
   {------------------------------------------------}
   {   errcode:= 3 : Matrix A not square            }
   {   errcode:= 4 : Vector RX has wrong dimension   }
   {   errcode:= 5 : Vector IX has wrong dimension   }
   {   errcode:= 6 : Matrix B has wrong dimension    }
   {------------------------------------------------}
   IF n <> ( UB(A,2) - LB(A,2) + 1 ) THEN errcode:= 3;
   IF n <>  ( UB(RX) - LB(RX) + 1 )  THEN errcode:= 4;
   IF n <>  ( UB(IX) - LB(IX) + 1 )  THEN errcode:= 5;
   IF n <> ( UB(B,1) - LB(B,1) + 1 ) THEN errcode:= 6;
   IF n <> ( UB(B,2) - LB(B,2) + 1 ) THEN errcode:= 6;

   IF errcode = 0 THEN
   BEGIN
      P:= 0.0;
      X:= RX;
      { find largest component in eigenvector approximation RX : }
      { keep index of this component in K0                       }
      FOR I:=  LB(A) TO UB(A) DO IF ABS(X[I])>P THEN BEGIN
                                                   P:= ABS(X[I]);
                                                   K0:= I;
                                                END;

      { K0-th unit vector : }
      VIID_K0    := 0.0;
      VIID_K0[K0]:= 1.0;

      { rescale approx. eigenvector such that largest component is = 1.0 }
      X    := (1.0/X[K0]) * X;

      READY:= TRUE;

      { compute approximate inverse : }
      X[K0]   :=  1.0;
      C       :=  #*( A - LAM*B );
      C[*,K0] :=  -B*X;
      MINV( C,ERR );
      READY:= ERR = 0;

      IF NOT ready THEN  errcode:= 2   { matrix probably singular }
      ELSE
      BEGIN
         { use K0-th component for computation of the eigenvalue }
         X[K0]   :=  LAM;

         { improve floating point approximation: }
         K:= 0;  bound:= 100.0*sqrt_01;
         REPEAT
            K:= K+1;

            { compute defect : }
            X[K0]:= 1.0;
            S :=  LAM*X;
```

```
   T :=  #*( LAM*X - S );
   RD:=  #*( A*X - B*S - B*T );
   X[K0]:= LAM;

   X0:= X;
   X := #*( X - C*RD );

   P := REL( X,X0 );

   LAM:= X[K0];

   bound:= bound * sqrt_01;
UNTIL (K>=2) AND ((K>=20) OR (P>=bound) OR (P<delta));


{ compute defect of improved approximation : }
X[K0]:= 1.0;

{ Decomposition of the product LAM*X into the sum RX1+RX2 }
RX1:= LAM*X;
RX2:= #*( LAM*X - RX1 );   { Attention: underflow possible ! }

FOR I:= lb(A) TO ub(A) DO
BEGIN
   { Decomposition of the term A*X-B*RX1-B*RX2 into the sum RD+D  }
   HD   :=  #( RVECTOR(A[I])*X
              - RVECTOR(B[I])*RX1 - RVECTOR(B[I])*RX2 );
   RD[I]:=  #*( HD );
   D [I]:=  ##( HD - RD[I] );
END;
D := - ##( C*RD + C*D );
Z :=   D;
X[K0]:= LAM;

{ residual :  E  -  R * funktional matrix }
FU := ##( A - LAM*B );
MB := ##( ID(A) - C*FU );

{ interval iteration : }
K    := 0;
eps  := 0.05;
READY:= FALSE;
X[K0]:= 1.0;
REPEAT
   K:= K+1;
   IF K>=5 THEN eps := 10.0*eps;

   ZA:= BLOW( Z, eps );

   { compute K0-th column of functional matrix : }
   ZA_K0   :=  ZA[K0];
   ZA[K0]  :=  0.0;
   FU[*,K0]:= -##( B*X + B*ZA );
   ZA[K0]  :=  ZA_K0;
   MB[*,K0]:=  ##( VIID_K0 - C*IVECTOR(FU[*,K0]) );

   Z:=  D + MB*ZA;
   READY:= Z IN ZA;

UNTIL READY OR (K>=10) OR TOO_BAD(Z);

ILAM := LAM + Z[K0];
Z[K0]:= 0.0;
```

```
        IX    := X + ORD(READY) * Z;
        errcode:= ORD( NOT READY );

      END { if not ready then ... else};

    END { errcode<>0 };

  END (* GEIG *);

  END.
```

3.2.5 Test Results

We demonstrate the use of the module `geig` by the following short PASCAL-XSC program which reads in a matrix A and eigenvalue and eigenvector approximations. Procedure `EIG` from the module `geig` is then used to enclose an eigenpair which is close to the given approximation. The approximations can be entered repeatedly until 'n' or 'N' is entered in procedure `main`. Then a new dimension n and another matrix A may be entered, otherwise the program stops (if $n \leq 0$ is entered).

```
PROGRAM geigtest;

USE i_ari,mv_ari,mvi_ari,geig;

PROCEDURE MAIN( N : INTEGER );

VAR ERR : INTEGER;
    A,B : RMATRIX[1..N,1..N];
    EV  : RVECTOR[1..N];
    EW  : REAL;
    IEV : IVECTOR[1..N];
    IEW : INTERVAL;
    OK  : BOOLEAN;
    C   : CHAR;

PROCEDURE WRITE_EW;
VAR J : INTEGER ;
BEGIN
   WRITELN;
   WRITELN('Eigenvalue  = ', IEW );
   WRITELN('Eigenvector = ', IEV );
END { WRITE_EW };

BEGIN
   WRITELN('A = '); READ(A);
   WRITELN('B = '); READ(B);

   REPEAT
      WRITE('Eigenvalue  approximation : ');  READ(EW);
      WRITE('Eigenvector approximation : ');  READ(EV);

      EIG( A,B,EW,EV,IEW,IEV,ERR );
      OK:= ERR=0;

      IF OK THEN  WRITE_EW  ELSE
      BEGIN
         WRITELN;
         WRITELN( 'Verification not succeeded. errcode = ', ERR );
      END;
```

```
      WRITELN;
      WRITE( 'another eigenvalue ? (y/n) ' );
      READLN; READ(C);

   UNTIL C IN ['N','n'];
END;

VAR N : INTEGER ;

BEGIN
   REPEAT
      WRITE(' N ? (End = 0) '); READ(N);
      IF N>0 THEN MAIN( N );
   UNTIL N<=0;
END.
```

As a first example we use the following two matrices A and B taken from [15] and [335].

$$A = \begin{pmatrix} 12 & 1 & -1 & 2 & 1 \\ 1 & 14 & 1 & -1 & 1 \\ -1 & 1 & 16 & -1 & 1 \\ 2 & -1 & -1 & 12 & -1 \\ 1 & 1 & 1 & -1 & 11 \end{pmatrix}, \quad B = \begin{pmatrix} 10 & 2 & 3 & 1 & 1 \\ 2 & 12 & 1 & 2 & 1 \\ 3 & 1 & 11 & 1 & -1 \\ 1 & 2 & 1 & 9 & 1 \\ 1 & 1 & -1 & 1 & 15 \end{pmatrix}$$

As approximation for λ we take $\tilde{\lambda} = 2.31$ and for the eigenvector $\tilde{x} = (-0.20, 0.093, 0.24, -0.17, 0.063)^T$.

The test program then gives the following output:

```
 N ? (End = 0) 5
A =
12  1 -1  2  1
 1 14  1 -1  1
-1  1 16 -1  1
 2 -1 -1 12 -1
 1  1  1 -1 11
B =
10  2  3  1  1
 2 12  1  2  1
 3  1 11  1 -1
 1  2  1  9  1
 1  1 -1  1 15
Eigenvalue  approximation : 2.31
Eigenvector approximation : -0.20 0.093 0.24 -0.17 0.063

Eigenvalue  = [  2.310604321348129E+000,  2.310604321348130E+000 ]
Eigenvector =
[ -8.523647246539458E-001, -8.523647246539456E-001 ]
[  3.881806704921011E-001,  3.881806704921012E-001 ]
[  1.000000000000000E+000,  1.000000000000000E+000 ]
[ -6.932489643679560E-001, -6.932489643679558E-001 ]
[  2.626493909653151E-001,  2.626493909653152E-001 ]

another eigenvalue ? (y/n) n
 N ? (End = 0) 0
```

This result agrees with the enclosures given in [15] where only a 13 digit decimal arithmetic was used.

As a second example we demonstrate that our algorithm also works in the case where the matrix B is singular. We take

$$A = \begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} .$$

Since B is singular the eigenvalue problem $Ax = \lambda Bx$ has only *one* eigenvalue $\lambda_1 = 3$ with the corresponding eigenvector $x_1 = (-2, 1)^T$. For the crude approximations $\tilde{\lambda}_1 = 3.5$ and $\tilde{x}_1 = (-2.5, 1.5)^T$ we get the following results:

```
 N ? (End = 0) 2
A =
1 2 0 3
B =
0 0 0 1
Eigenvalue  approximation : 3.5
Eigenvector approximation : -2.5 1.5

Eigenvalue  = [  2.999999999999999E+000,  3.000000000000001E+000 ]
Eigenvector =
[  1.000000000000000E+000,  1.000000000000000E+000 ]
[ -5.000000000000000E-001, -5.000000000000000E-001 ]

another eigenvalue ? (y/n) n
 N ? (End = 0) 0
```

Even if both matrices, A and B are singular the algorithm still works, as can be seen from this final example:

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} .$$

which has the only eigenvalue $\lambda_1 = 0$ with the corresponding eigenvector $x_1 = (0, 1)^T$. For $\tilde{\lambda}_1 = 0.5$ and $\tilde{x}_1 = (0.5, 1.5)^T$ we get the output:

```
 N ? (End = 0) 2
A =
1 0 0 0
B =
0 0 0 1
Eigenvalue  approximation : 0.5
Eigenvector approximation : 0.5 1.5

Eigenvalue  = [  0.000000000000000E+000,  0.000000000000000E+000 ]
Eigenvector =
[  0.000000000000000E+000,  0.000000000000000E+000 ]
[  1.000000000000000E+000,  1.000000000000000E+000 ]

another eigenvalue ? (y/n) n
 N ? (End = 0) 0
```

Thus the program gets the results even *exactly* in this case.

3.3 The Symmetric Eigenvalue Problem $Ax = \lambda x, A^T = A$

In this section we present a method which computes highly accurate enclosures for the eigenvalues of a symmetric real matrix $A = A^T$. This method consists of three basic steps. First we compute eigenvalue and eigenvector approximations (by the classical Jacobi method in our case). Second, using the matrix T of the eigenvector approximations we compute an enclosure of the transformed matrix $A_1 = T^{-1}AT$ and third, for A_1 we use Gerschgorin's circle theorem to derive enclosing intervals for the eigenvalues. The eigenvector then can be enclosed by standard estimations, see e.g. [333]. Increased accuracy will be obtained by additional measures like a modification of Jacobi's method, reorthonormalizations of the eigenvector approximations and the use of the exact scalar product.

3.3.1 Introduction

It is well known that for the eigenvalue problem with a real, symmetric matrix

$$Ax = \lambda x, \quad A^X = A, \tag{3.14}$$

there exists a real orthogonal matrix X (i.e. $X = X^H$) which transforms A into a diagonal matrix D

$$D = X^{-1}AX \; .$$

The columns of X are the eigenvectors and the diagonal elements of D are the eigenvalues of A.

Therefore, if $T \approx X$ is an approximation of the eigensystem, then

$$A_1 = T^{-1}AT \; . \tag{3.15}$$

will be a nearly or strongly diagonal dominant matrix, depending on the quality of the approximation T.

We will use such an approximation to compute an enclosure $[A_1]$ of A_1 and then we will apply Gerschgorin's circle theorem to $[A_1]$ to obtain safe enclosures of the eigenvalues of A.

The matrix T may be computed by the well known Jacobi method, see [309] or [308]. For details on the enclosure method presented in this section see Lohner, [231].

3.3.2 Theoretical Background

Assume we have an approximate eigensystem T of A which was computed e.g. by the Jacobi method. Then we can get an enclosure of the transformed matrix A_1 from (3.15) by two different approaches.

First we can rewrite (3.15) as a linear matrix equation

$$TA_1 = AT$$

and can compute an enclosure for A_1 by the methods presented in Chapter 2. Here the product AT has to be enclosed by an interval matrix, or the methods of Chapter 2 can be slightly modified to incorporate this product as a part of the enclosure method (i.e. to allow scalar products in the right hand side of the linear system).

Second, since T is almost orthogonal we can again use the same procedure as in the case of the coordinate transformation with orthogonal matrices in Section 2.2.2.1. An enclosure $[T^{-1}]$ of T^{-1} can thus be computed from (2.31) and we get

$$[A_1] = [T^{-1}](AT) .$$

Now, if T was a sufficiently good approximate eigensystem, then $[A_1]$ is strongly diagonally dominant and we can apply Gerschgorin's circle theorem, [309], to obtain enclosures of the eigenvalues:

$$\lambda_i \in [\underline{a}_{ii} - \sum_{\substack{j=1\\ i\neq j}}^{n} |[a_{ij}]|, \overline{a}_{ii} + \sum_{\substack{j=1\\ i\neq j}}^{n} |[a_{ij}]|] .$$

The eigenvectors then can be enclosed by use of a standard estimation from Wilkinson, [333]. Let λ be an exact eigenvalue and $\tilde{\lambda}$ its approximation. Furthermore let x be the corresponding exact eigenvector, $\tilde{x}$ an approximation and both are assumed to be normalized in the Euklidian norm, $||x||_2 = ||\tilde{x}||_2 = 1$ with the residual norm $\epsilon = ||A\tilde{x} - \tilde{\lambda}\tilde{x}||_2$. Then

$$||x - \tilde{x}||_2^2 \leq \frac{\epsilon^2}{a^2} + \left[1 - (1 - \frac{\epsilon^2}{a^2})^{1/2}\right]^2 \tag{3.16}$$

where a is the largest distance of $\tilde{\lambda}$ to the next eigenvalue, i.e. λ and its approximation $\tilde{\lambda}$ are contained in $[\tilde{\lambda} - a, \tilde{\lambda} + a]$ and all the other eigenvalues are outside of this interval (and a should be chosen as large as possible, of course).

The PASCAL-XSC program in the following section does not compute eigenvector enclosures from this estimation. However, it computes further refinements of the eigenvalue enclosures. More details on these refinements can be found in Lohner, [231].

3.3.3 PASCAL-XSC Program Code

```
MODULE seig;

use i_ari,mv_ari,mvi_ari;


GLOBAL PROCEDURE SEIG( VAR A0 : RMATRIX;
                       VAR EW : IVECTOR; VAR ERRCODE : INTEGER );
{ SEIG computes enclosures for all eigenvalues of the symmetric matrix A0  }
{ A0 is the symmetric matrix (input parameter), symmetry is not checked !  }
```

```
{ EW is the output vector of the eigenvalue enclosures (in unsorted order) }
{ ERRCODE is an error indicator (always = 0 in this version of SEIG)       }

VAR  HM,A,EV,EV1,R  : RMATRIX[1..UB(A0),1..UB(A0)];
     D,HV           : RVECTOR[1..UB(A0)];
     AJ             : IVECTOR[1..UB(A0)];
     DP             : DOTPRECISION;
     NRM,S,
     EPS1,EPS2,EPS3 : REAL;
     I,J,N          : INTEGER;

PROCEDURE ORTHONORM( VAR Q : RMATRIX );
{ Gram-Schmidt orthogonalization of the columns of Q }
VAR J,K : INTEGER;
    R   : RVECTOR[1..N];
BEGIN
   R:= 0.0;
   FOR K:= 1 TO N DO
   BEGIN
      { orthogonalize column k : }
      FOR J := 1 TO K-1 DO  R[J]:= -rvector(q[*,j])*rvector(q[*,k]) ;
      R[K]  := 1.0;
      Q[*,K]:= Q*R;
      { norm column k to length 1 : }
      Q[*,K]:= (1/SQRT( rvector(Q[*,K])*rvector(Q[*,K]) )) * RVECTOR(Q[*,K]);
   END;
END;

PROCEDURE ORTHONORM( VAR Q,Q1 : RMATRIX );
{ Gram-Schmidt orthogonalization of the columns of Q + Q1 }
VAR J,K            : INTEGER;
    A,B,QK,Q1K,R : RVECTOR[1..N];
    Z,U,V          : REAL;
    DP             : DOTPRECISION;
BEGIN
   R:= 0.0;
   FOR K:= 1 TO N DO
   BEGIN
      { orthogonalize column k : }
      FOR J:= 1 TO K-1 DO
         R[J]:= #*(   RVECTOR( Q[*,J])*RVECTOR( Q[*,K])
                    + RVECTOR( Q[*,J])*RVECTOR(Q1[*,K])
                    + RVECTOR(Q1[*,J])*RVECTOR( Q[*,K])
                    + RVECTOR(Q1[*,J])*RVECTOR(Q1[*,K])  );
      R:= -R;
      R[K]:= 1.0;
      QK := #*( Q*R + Q1*R );
      Q1K:= #*( Q*R + Q1*R - QK );
      { norm column k to length 1 : }
      DP:= #(  QK*QK + QK*Q1K + Q1K*QK + Q1K*Q1K );
      Z:= #*( DP );
      U:= SQRT(Z);
      Z:= #*( DP - Z );
      V:= #*( DP - U*U );
      V:= V/(2*U);        { U + V  =  SQRT(DP)   }
      Z:= 1/(U+V);
      A:= Z*( QK + Q1K );
      B:= #*( QK + Q1K - U*A - V*A );
      Q1[*,K]:= Z*B;
      Q [*,K]:=  A;
   END;
END;

FUNCTION TRANSFORM( VAR D: RVECTOR; VAR A,T: RMATRIX )
```

```
                   : RMATRIX[1..N,1..N];
VAR J   : INTEGER;
    B   : RMATRIX[1..N,1..N];
BEGIN
   B:= 0.0;
   FOR J:= 1 TO N DO  B[J,J]:= D[J];
   B:= #*( A*T - T*B );
   TRANSFORM:= TRANSP(T)*B;
END;

FUNCTION TRANSFORM( VAR D: RVECTOR; VAR A,T,T1: RMATRIX )
                   : RMATRIX[1..N,1..N];
VAR J : INTEGER;
    B : RMATRIX[1..N,1..N];
BEGIN
   B:= 0.0;
   FOR J:= 1 TO N DO  B[J,J]:= D[J];
   B:= #*( A*T + A*T1 - T*B - T1*B );
   TRANSFORM:= #*( TRANSP(T)*B + TRANSP(T1)*B );
END;

FUNCTION TRANSFORM( VAR D : RVECTOR; VAR A,T,T1 : RMATRIX;
                        J : INTEGER; EPS : INTERVAL ) : IVECTOR[1..N];
VAR I,K    : INTEGER;
    B,C    : IVECTOR[1..N];
    H,H1   : RVECTOR[1..N];
BEGIN
   H := RVECTOR(T [*,J]);
   H1:= RVECTOR(T1[*,J]);
   C := ##( A*H + A*H1 - D[J]*H - D[J]*H1 );
   FOR I:= 1 TO N DO
   BEGIN
      FOR K:=1 TO N DO B[K]:= T[K,I] + ( T1[K,I] + EPS );
      TRANSFORM[I]:= B*C;
   END;
END;

FUNCTION NORM( VAR Q,Q1 : RMATRIX ) : REAL;
VAR I,J        : INTEGER;
    MAX,Y,S    : REAL;
    D          : INTERVAL;
    G,G1,H,H1  : RVECTOR [1..N];
BEGIN
   MAX:= 0.0;
   FOR I:= 1 TO N DO
   BEGIN
      S:= 0.0;
      G := RVECTOR(Q [*,I]);
      G1:= RVECTOR(Q1[*,I]);
      FOR J:= 1 TO N DO
      BEGIN
         Y := ORD( I=J );
         H := RVECTOR(Q [*,J]);
         H1:= RVECTOR(Q1[*,J]);
         D := ##( Y - G*H - G*H1 - G1*H - G1*H1 );
         S := S + SUP(ABS(D));
      END;
      IF S>MAX THEN MAX:= S;
   END;
   NORM:= MAX;
END;

PROCEDURE JACOBI( VAR D : RVECTOR; VAR A,W : RMATRIX;
                  LEVEL : INTEGER; EPS : REAL ) ;
```

```
VAR I,J,K     : INTEGER;
    AJ,AK,AH : RVECTOR[1..N];
    DP       : DOTPRECISION;
    READY    : BOOLEAN;
    C,S,T,TH,TAU,
    MAX,MIND : REAL;
BEGIN
   I:= 0;
   REPEAT
      I:= I+1;
      FOR J:=  1  TO N-1 DO
      BEGIN
         DP    := #( D[J] + A[J,J] );
         D[J]  := #*( DP );
         A[J,J]:= #*( DP - D[J] );
         FOR K:= J+1 TO  N  DO
         IF A[J,K]<>0.0 THEN
         BEGIN
            TH:= #*( D[J] + A[J,J] - D[K] - A[K,K] );
            T := 2*A[J,K];
            IF TH=0.0 THEN T:= 1
                      ELSE T:= T / ( ABS(TH) + SQRT(SQR(T)+SQR(TH)) );
            IF TH<0.0 THEN T:= -T;
            C    := 1/SQRT(1+SQR(T));
            S    := T*C;
            TAU  := S/(1+C);

            AJ:= RVECTOR(A[*,J]);
            AK:= RVECTOR(A[*,K]);
            AH:= AJ + S*( AK-TAU*AJ );
            AK:= AK - S*( AJ+TAU*AK );
            AJ:= AH;

            DP   := #( D[J] + A[J,J] + T*A[J,K] );
            D[J] := #*( DP );
            AJ[J]:= #*( DP - D[J] );

            DP   := #( D[K] + A[K,K] - T*A[J,K] );
            D[K] := #*( DP );
            AK[K]:= #*( DP - D[K] );

            AJ[K]:= 0.0;
            AK[J]:= 0.0;

            A[J]:= AJ; A[*,J]:= AJ;
            A[K]:= AK; A[*,K]:= AK;

            AJ:= RVECTOR(W[*,J]);
            AK:= RVECTOR(W[*,K]);
            IF LEVEL=0 THEN
            BEGIN
               W[*,J]:= C*AJ + S*AK;
               W[*,K]:= C*AK - S*AJ;
            END ELSE
            BEGIN
               AH:= AJ + S*( AK - TAU*AJ );
               AK:= AK - S*( AJ + TAU*AK );
               AJ:= AH;
               AJ[J]:= AJ[J] - S*TAU;
               AJ[K]:= AJ[K] + S;
               AK[J]:= AK[J] - S;
               AK[K]:= AK[K] - S*TAU;
               W[*,J]:= AJ;
               W[*,K]:= AK;
```

```
            END;
         END; { FOR K }
      END; { FOR J }

      READY:= TRUE;
      FOR J:= 1 TO N DO
      BEGIN
         MIND:= ABS( D[J] );
         MAX:= 0.0;
         FOR K:= 1 TO J-1 DO
         BEGIN
            IF ABS(A[J,K])>MAX  THEN  MAX:= ABS(A[J,K]);
            IF ABS(A[K,J])>MAX  THEN  MAX:= ABS(A[K,J]);
         END;
         READY:= READY AND ( MAX <= MIND*EPS );
      END;
   UNTIL READY;
END;

BEGIN { SEIG }
   EPS1:= 1E-3;
   EPS2:= 1E-5;
   EPS3:= 1E-14;
   N   := UB(A0);

   IF ub(A,2) <> N THEN errcode:= 1 { matrix A0 not square }
   ELSE
   BEGIN
      EV  := ID(N);
      EV1:= 0.0;
      D   := 0.0;
      R   := 0.0;

      A:= A0;

      JACOBI( D, A, EV, 0, EPS1 );
      ORTHONORM( EV );
      A:= TRANSFORM( D, A0, EV );

      JACOBI( D, A, EV, 0, EPS2 );
      ORTHONORM( EV, R );
      A:= TRANSFORM( D, A0, EV, R );

      JACOBI( D, A, EV1, 1, EPS3/N );

      FOR J:= 1 TO N DO
      BEGIN
         FOR I:= 1 TO N DO
         BEGIN
            DP:= #( EV[I,J] + R[I,J] + RVECTOR(EV[I])*RVECTOR(EV1[*,J])
                                     + RVECTOR( R[I])*RVECTOR(EV1[*,J]) );
            HM[I,J]:= #*( DP );
            HV[I]  := #*( DP - HM[I,J] );
         END;
         EV1[*,J]:= HV;
      END;
      EV:= HM;

      ORTHONORM( EV, EV1 );

      NRM:= NORM( EV, EV1 );
      NRM:= NRM /> ( 1.0 -< NRM );

      FOR J:= 1 TO N DO
```

```
      BEGIN
         AJ:= TRANSFORM( D, A0, EV, EV1, J, INTVAL(-NRM,NRM) );

         S:= 0.0;
         FOR I:= 1 TO N DO   IF I<>J THEN S:= S +> SUP(ABS(AJ[I]));

         EW[J]:= INTVAL( D[J] +< ( AJ[J].INF -< S ),
                         D[J] +> ( AJ[J].SUP +> S )  );

      END;

      ERRCODE:= 0;

   END { errcode=0 };

END  { SEIG };

END.
```

3.3.4 Test Results

We demonstrate the use of the module `seig` by the following short PASCAL-XSC program which computes the eigenvalues of a modified Zielke matrix. This $n \times n$-matrix A depends on three parameters a, b and c and has the form:

$$A = \begin{pmatrix} a+b & a+b & \cdot & \cdot & a+b & a \\ a+b & \cdot & \cdot & a+b & a & \cdot \\ \cdot & \cdot & \cdot & a & \cdot & \cdot \\ \cdot & a+b & \cdot & \cdot & \cdot & \cdot \\ a+b & a & \cdot & \cdot & \cdot & a \\ a & \cdot & \cdot & \cdot & a & a-c \end{pmatrix}.$$

If b and c are very small compared to a then A has one large eigenvalue $\lambda_1 \approx na$ and the other ones are clustered in the order of magnitude of b and c. If $b = c$, however, then one of the eigenvalues becomes extremely small as can be seen from the output of the following PASCAL-XSC program:

```
PROGRAM seigtest;

USE i_ari,mv_ari,mvi_ari,seig;

PROCEDURE MAIN( N : INTEGER );

VAR ERR,i,j  : INTEGER;
    a,b,c    : REAL;
    A0       : RMATRIX[1..N,1..N];
    IEW      : IVECTOR[1..N];
BEGIN
   WRITE('a,b,c = '); READ(a,b,c);
   FOR i:= 1 TO N DO
   FOR j:= 1 TO N DO
       IF i+j <= N THEN A0[i,j]:= a + b
                   ELSE A0[i,j]:= a;
   A0[N,N]:= a - c;

   SEIG( A0,IEW,ERR );
```

```
    WRITELN('Eigenvalues :');
    FOR i:= 1 TO N DO WRITELN( i:3, ': ', IEW[i] );
END;

VAR N : INTEGER ;

BEGIN
    WRITELN('Eigenvalues of a modified Zielke matrix:');
    REPEAT
        WRITE(' N ? (End = 0) '); READ(N);
        IF N>0 THEN MAIN( N );
    UNTIL N<=0;
END.
```

If we choose $n = 4, a = 1E12, b = c = 1$ then we get the output:

```
Eigenvalues of a modified Zielke matrix:
 N ? (End = 0) 4
a,b,c = 1e12 1 1
Eigenvalues :
  1: [  4.000000000001250E+012,  4.000000000001251E+012 ]
  2: [  5.930703308172123E-001,  5.930703308172125E-001 ]
  3: [ -8.430703308172594E-001, -8.430703308172592E-001 ]
  4: [         -5.0000003E-013,         -4.9999997E-013 ]
 N ? (End = 0) 0
```

whereas for $n = 4, a = 1E12, b = 1, c = 0$ the smalles eigenvalues disappears:

```
Eigenvalues of a modified Zielke matrix:
a,b,c = 1e12 1 0
Eigenvalues :
  1: [  4.000000000001500E+012,  4.000000000001501E+012 ]
  2: [  7.071067811863340E-001,  7.071067811863342E-001 ]
  3: [ -7.071067811865843E-001, -7.071067811865840E-001 ]
  4: [  4.999999999999374E-001,  4.999999999999376E-001 ]
 N ? (End = 0) 0
```

The same occurs for higher dimensions of A: with $n = 12$ and the two parameter combinations as above we get:

```
Eigenvalues of a modified Zielke matrix:
 N ? (End = 0) 12
a,b,c = 1e12 1 1
Eigenvalues :
  1: [  1.200000000000541E+013,  1.200000000000542E+013 ]
  2: [ -7.467361729439217E-001, -7.467361729439215E-001 ]
  3: [  1.866930882952880E+000,  1.866930882952881E+000 ]
  4: [  9.504870176907925E-001,  9.504870176907927E-001 ]
  5: [  5.053846930161961E-001,  5.053846930161963E-001 ]
  6: [  6.701945184379756E-001,  6.701945184379758E-001 ]
  7: [ -2.026244464896115E+000, -2.026244464896113E+000 ]
  8: [ -5.924967207430753E-001, -5.924967207430751E-001 ]
  9: [ -1.075275252928013E+000, -1.075275252928012E+000 ]
 10: [ -5.210247170381131E-001, -5.210247170381129E-001 ]
 11: [  5.521135497841489E-001,  5.521135497841491E-001 ]
 12: [          -5.000001E-013,          -4.999999E-013 ]
 N ? (End = 0) 12
a,b,c = 1e12 1 0
Eigenvalues :
  1: [  1.200000000000550E+013,  1.200000000000551E+013 ]
  2: [ -7.071067811865598E-001, -7.071067811865596E-001 ]
  3: [  1.931851652577744E+000,  1.931851652577746E+000 ]
  4: [  9.999999999998749E-001,  9.999999999998751E-001 ]
```

```
  5: [  5.176380902049976E-001,  5.176380902049978E-001 ]
  6: [  5.773502691895738E-001,  5.773502691895740E-001 ]
  7: [ -1.931851652578368E+000, -1.931851652578366E+000 ]
  8: [ -5.773502691896296E-001, -5.773502691896293E-001 ]
  9: [ -1.000000000000042E+000, -1.000000000000041E+000 ]
 10: [ -5.176380902050424E-001, -5.176380902050422E-001 ]
 11: [  4.999999999999791E-001,  4.999999999999792E-001 ]
 12: [  7.071067811864762E-001,  7.071067811864765E-001 ]
 N ? (End = 0) 0
```

It is amazing that in the case $c = 0$ the eigenvalues are almost symmetric to zero (all but the smallest and biggest one) but this symmetry is disturbed in the last few digits.

In the previous examples the matrices A could be represented exactly in the binary floating-point system of the computer since they had only integer entries. We now modify these matrices by multiplying them with a factor of 10^{-12}. Then of course the eigenvalues will be multiplied by the same factor, i.e. they will be unchanged in their (decimal) mantissa, only the exponent is decreased by 12.

However, these modified matrices are no longer representable exactly in the computer. Thus the program will compute eigenvalues of slightly perturbed matrices, the perturbation resulting *only* from rounding the input matrices to the floating-point screen. Still, however, this can have dramatic effects in the change of the eigenvalues. The results are contained in the following output:

```
Eigenvalues of a modified Zielke matrix:
 N ? (End = 0) 4
a,b,c = 1 1e-12 1e-12
Eigenvalues :
  1: [  4.000000000001249E+000,  4.000000000001251E+000 ]
  2: [ -8.431193251503541E-013, -8.431193251503539E-013 ]
  3: [  5.931248540398159E-013,  5.931248540398161E-013 ]
  4: [  5.551269125267447E-017,  5.551269125267455E-017 ]
 N ? (End = 0) 4
a,b,c = 1 1e-12 0
Eigenvalues :
  1: [  4.000000000001500E+000,  4.000000000001502E+000 ]
  2: [ -7.071696433912090E-013, -7.071696433912088E-013 ]
  3: [  5.000444502911079E-013,  5.000444502911081E-013 ]
  4: [  7.071696433909587E-013,  7.071696433909589E-013 ]
 N ? (End = 0) 12
a,b,c = 1 1e-12 1e-12
Eigenvalues :
  1: [  1.200000000000541E+001,  1.200000000000542E+001 ]
  2: [  1.867101224492465E-012,  1.867101224492467E-012 ]
  3: [   5.55126912526744E-017,   5.55126912526746E-017 ]
  4: [ -5.925472500549965E-013, -5.925472500549963E-013 ]
  5: [  5.521628977880098E-013,  5.521628977880100E-013 ]
  6: [  5.054296497731113E-013,  5.054296497731115E-013 ]
  7: [ -2.026408947546140E-012, -2.026408947546139E-012 ]
  8: [ -5.210705968225934E-013, -5.210705968225932E-013 ]
  9: [ -7.467960453734009E-013, -7.467960453734007E-013 ]
 10: [  9.505735015590511E-013,  9.505735015590514E-013 ]
 11: [  6.702549300436453E-013,  6.702549300436455E-013 ]
 12: [ -1.075356814683535E-012, -1.075356814683534E-012 ]
 N ? (End = 0) 12
a,b,c = 1 1e-12 0
Eigenvalues :
  1: [  1.200000000000549E+001,  1.200000000000551E+001 ]
```

```
  2: [  1.932023395314655E-012,  1.932023395314656E-012 ]
  3: [ -7.071696433911846E-013, -7.071696433911844E-013 ]
  4: [ -5.176841085327035E-013, -5.176841085327033E-013 ]
  5: [  5.774015959647195E-013,  5.774015959647198E-013 ]
  6: [  5.000444502911496E-013,  5.000444502911498E-013 ]
  7: [ -1.932023395315278E-012, -1.932023395315277E-012 ]
  8: [ -5.774015959647753E-013, -5.774015959647751E-013 ]
  9: [ -1.000088900582383E-012, -1.000088900582382E-012 ]
 10: [  1.000088900582215E-012,  1.000088900582217E-012 ]
 11: [  7.071696433911010E-013,  7.071696433911012E-013 ]
 12: [  5.176841085326586E-013,  5.176841085326588E-013 ]
N ? (End = 0) 0
```

Chapter 4

Evaluation of Polynomials in Several Variables

A widely used method for the numerical computation of the value of a polynomial in several variables is the nested Horner scheme. It is well known that even in the case of a polynomial in one variable, the computed values using Horner's scheme may be far from the correct values of the polynomial. This is especially true for polynomials in several variables.

In case of a polynomial in two variables

$$p(x,y) = \sum_{j=0}^{m}\sum_{i=0}^{n} a_{ij}x^i y^j = \sum_{j=0}^{m}(\sum_{i=0}^{n} a_{ij}x^i)y^j = \sum_{i=0}^{n}(\sum_{j=0}^{m} a_{ij}y^j)x^i$$

two variants of the nested Horner scheme, which are mathematically equivalent, are obvious. The first variant interprets $p(x,y)$ as a one-dimensional polynomial in y, the coefficients of which are one-dimensional polynomials in x. In the second version, $p(x,y)$ is considered to be a one-dimensional polynomial in x with coefficients which are one-dimensional polynomials in y. For

$$p(x,y) = x^6 + 3x^5y - 5x^3y^3 + 3xy^5 - y^6 \tag{4.1}$$

this leads to

$$p(x,y) = (((-y+3x)y^2 - 5x^3)y^2 + 3x^5)y + x^6 \tag{4.2}$$

as the first variant and to

$$p(x,y) = (((x+3y)x^2 - 5y^3)x^2 + 3y^5)x - y^6 \tag{4.3}$$

as the second. Using a hexadecimal arithmetic with 14 mantissa digits (IBM\370), one obtains

$$\tilde{p}(4181, 6765) \quad = \quad 1048576$$

in case (4.2) and

$$\tilde{p}(4181, 6765) \quad = \quad 0$$

in case (4.3). $\tilde{p}$ means the machine evaluation of p. Indeed the correct value is $p(4181, 6765) = 1$.

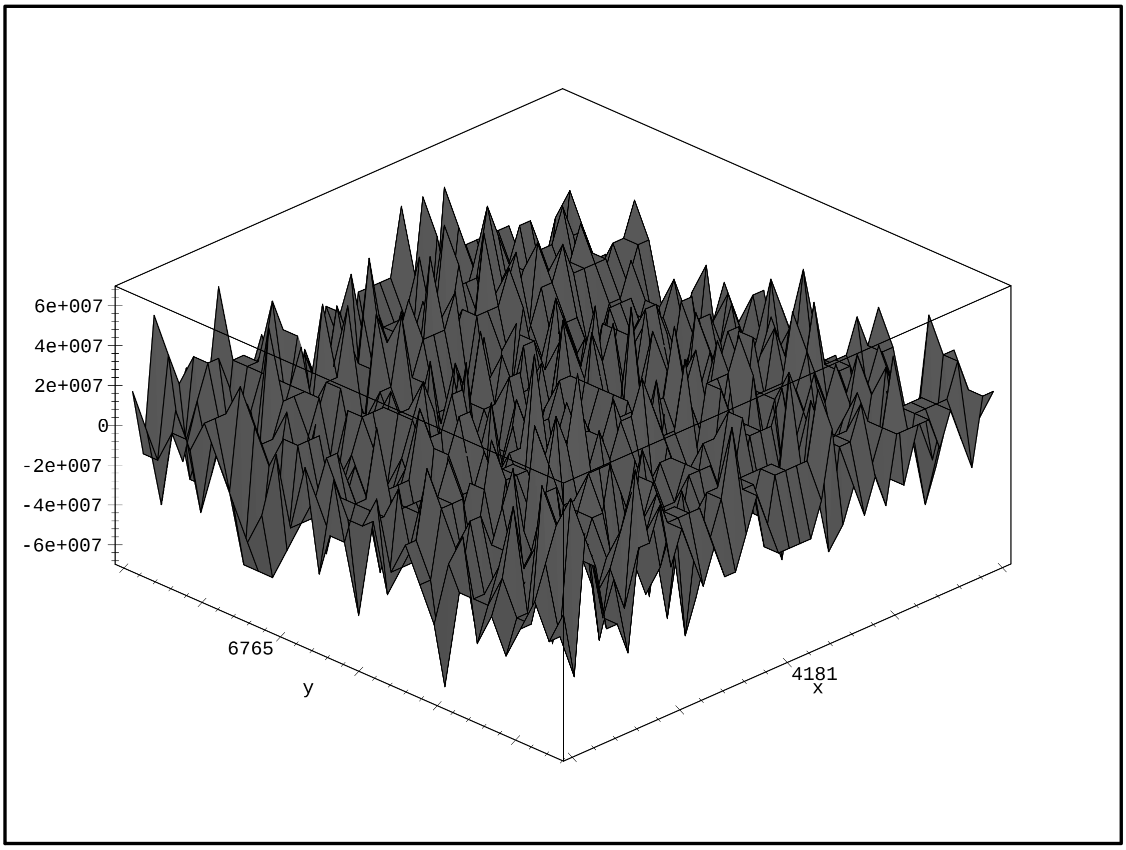


Figure 4.1: Rounding Errors in the Computation of Polynomial (4.1)

The question arises how such catastrophic errors may be overcome. The algorithm of Böhm [60, 61] to compute the value of one-dimensional polynomials with maximum accuracy is based on a transformation of the polynomial into an equivalent system of linear equations. Such a system can be solved with maximum accuracy using the residual iteration technique. To compute the value of a polynomial in several variables, a modified version of the algorithm given by Böhm is used by Lohner [230]. This algorithm does not only allow floating-point, but also so-called staggered interval coefficients (a sort of multiple-precision intervals) [190, 233]. The desired value of the (one-dimensional) polynomial is included in a staggered interval with a required accuracy. This algorithm is used repeatedly to compute intermediate results with increased accuracy as they occur in the nested Horner's scheme. The specification of an appropriate accuracy for the staggered formats at each level of the nested scheme leads to a final result with high accuracy.

4.1 Mathematical Background

In the following a different approach is used. The original idea of Böhm is the transformation of a one-dimensional polynomial into an equivalent system of linear

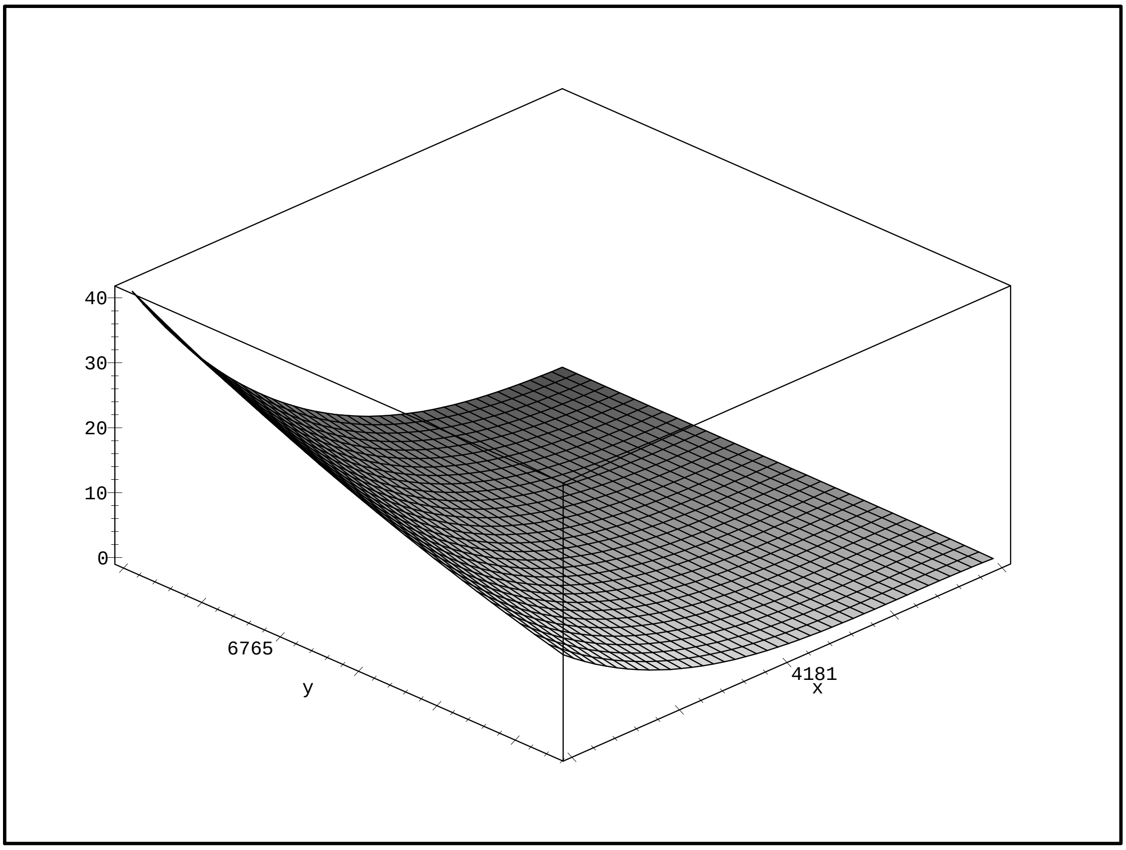


Figure 4.2: Polynomial (4.1) Without Rounding Errors

equations by introducing auxiliary variables for all intermediate results occurring in Horner's scheme. The same steps may be performed for polynomials in several variables [192]. This is illustrated by the following example. The polynomial

$$\begin{aligned} p(x,y) &= \sum_{j=0}^{3}(\sum_{i=0}^{2} a_{ij}x^i)y^j \\ &= \{\ \{\ \overbrace{[(a_{23}x + a_{13})x + a_{03}]}^{b_3(x)} \cdot y \quad + \quad \overbrace{[(a_{22}x + a_{12})x + a_{02}]}^{b_2(x)}\} \cdot y \\ &\qquad \overbrace{[(a_{21}x + a_{11})x + a_{01}]}^{b_1(x)}\} \cdot y \quad + \quad \overbrace{[(a_{20}x + a_{10})x + a_{00}]}^{b_0(x)} \end{aligned} \tag{4.4}$$

is considered. Using auxiliary variables x_i for all intermediate results, one obtains

$$\left.\begin{aligned}
x_1 &= a_{23}\\
x_2 &= x_1 \cdot x + a_{13}\\
x_3 &= x_2 \cdot x + a_{03}\\
x_4 &= a_{22}\\
x_5 &= x_4 \cdot x + a_{12}\\
x_6 &= x_3 \cdot y + x_5 \cdot x + a_{02}\\
x_7 &= a_{21}\\
x_8 &= x_7 \cdot x + a_{11}\\
x_9 &= x_6 \cdot y + x_8 \cdot x + a_{01}\\
x_{10} &= a_{20}\\
x_{11} &= x_{10} \cdot x + a_{10}\\
x_{12} &= x_9 \cdot y + x_{11} \cdot x + a_{00}\\
&= p(x,y)
\end{aligned}\right\}\text{equivalently}\left\{\begin{aligned}
x_1 &= a_{23}\\
-x_1 \cdot x + x_2 &= a_{13}\\
-x_2 \cdot x + x_3 &= a_{03}\\
x_4 &= a_{22}\\
-x_4 \cdot x + x_5 &= a_{12}\\
-x_3 \cdot y - x_5 \cdot x + x_6 &= a_{02}\\
x_7 &= a_{21}\\
-x_7 \cdot x + x_8 &= a_{11}\\
-x_6 \cdot y - x_8 \cdot x + x_9 &= a_{01}\\
x_{10} &= a_{20}\\
-x_{10} \cdot x + x_{11} &= a_{10}\\
-x_9 \cdot y - x_{11} \cdot x + x_{12} &= a_{00}
\end{aligned}\right.$$

4.2 The Problem in Matrix-Vector Notation

In matrix-vector notation, this means that a system $A \cdot x = b$ must be solved. In the general case of a polynomial of degree n in x and degree m in y , one obtains

$$\underbrace{\begin{pmatrix} B & & & & \\ -Y & B & & & 0 \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ 0 & & & -Y & B \end{pmatrix}}_{A} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{dim} \end{pmatrix} = \underbrace{\begin{pmatrix} a_{n\,m} \\ a_{n-1\,m} \\ \vdots \\ \vdots \\ a_{0\,0} \end{pmatrix}}_{b}$$

with

$$B = \underbrace{\begin{pmatrix} 1 & & & & \\ -x & 1 & & & 0 \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ 0 & & & -x & 1 \end{pmatrix}}_{n+1} \quad \text{and} \quad Y = \underbrace{\begin{pmatrix} 0 & \dots & & 0 \\ \vdots & 0 & & \vdots \\ \vdots & & 0 & 0 \\ 0 & \dots & 0 & y \end{pmatrix}}_{n+1}.$$

This is a system of linear equations with a very special lower triangular sparse matrix of dimension $dim = (m+1) \cdot (n+1)$. The last component of the solution vector is the value of the polynomial. The diagonal elements of the matrix A are all 1 and the off-diagonal elements are the components of the point at which the polynomial is to be evaluated, with negative sign. The matrix does not depend on the polynomial coefficients. It is determined by the number of variables of the polynomial along with the maximum degree of each of these variables, and by the point at which the polynomial is evaluated. The right-hand side b of the system is given by the coefficients of the polynomial.

Let $A^{(r-1)} =: D$ be the matrix corresponding to the $r-1$ dimensional polynomial $p(t_1, t_2, \cdots, t_{r-1})$. The system corresponding to the r-dimensional problem is then given by

$$\underbrace{\begin{pmatrix} D & & & & \\ -X_r & D & & & \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ & & & -X_r & D \end{pmatrix}}_{A^{(r)}} \cdot \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_{dim} \end{pmatrix} = \underbrace{\begin{pmatrix} a_{n_1 n_2 \dots n_r} \\ \vdots \\ \vdots \\ \vdots \\ a_{00\dots0} \end{pmatrix}}_{b^{(r)}} \quad . \tag{4.5}$$

The sparse lower triangular matrix $A^{(r)}$ corresponding to the polynomial

$$p(t_1, t_2, \cdots, t_r) = \sum_{i_r=0}^{n_r} \cdots \sum_{i_2=0}^{n_2} \sum_{i_1=0}^{n_1} a_{i_1, i_2, \cdots, i_r} t_1^{i_1} t_2^{i_2} \cdots t_r^{i_r} \tag{4.6}$$

consists of

$$dim = \prod_{k=1}^{r} (n_k + 1)$$

rows. Only the values $1,\ -t_1,\ -t_2,\ \cdots,\ -t_r$ appear as elements of $A^{(r)}$. Again the right hand side $b^{(r)}$ is given by the coefficients of the polynomial $a_{i_1, i_2, \cdots, i_r}$. They are ordered in such a way that the indices to the left are decreasing more rapidly. The value of the last component x_{dim} of the solution x of the system is the value $p(t_1, t_2, \cdots, t_r) = x_{dim}$ of the polynomial. X_r is the $(n_{r-1}+1)(n_{r-2}+1) \cdots (n_1+1)$ dimensional matrix

$$X_r = \begin{pmatrix} 0 & \cdots & 0 & 0 \\ \vdots & & \vdots & \vdots \\ & & & 0 \\ 0 & \cdots & 0 & t_r \end{pmatrix} \quad \text{, with } t_r \text{ being the rightmost variable.}$$

The structure of the matrix $A^{(r)}$ may be expressed as follows: If a variable x_k appears in some row of $A^{(r)}$, the distance $dist_k$ from the diagonal element of this row is always the same constant value given by:

$$\left.\begin{aligned} dist_1 &:= 1, \\ dist_{k+1} &:= dist_k \cdot (n_k + 1), \quad k = 1, 2, \dots, r \ . \end{aligned}\right\} \tag{4.7}$$

All diagonal elements of $A^{(r)}$ are 1. The value $-t_k$ appears in the i−th row of the matrix $A^{(r)}$ if and only if

$$i \equiv 0 \bmod dist_k \quad \text{and} \quad i \not\equiv dist_k \bmod dist_{k+1} \tag{4.8}$$

Here, the indices vary in the ranges $i = 1, 2, \ldots, dim$, $\quad k = 1, 2, \ldots, r$. All other elements of the sparse matrix are 0.

The whole information of the problem is encoded in this system of linear equations. If the point of evaluation and the original coefficients of the polynomial are floating-point numbers, then the system is exactly representable on the computer.

The following figure shows the structure of the matrix corresponding to a three-dimensional polynomial

$$
\begin{aligned}
p(x, y, z) \quad = \quad & x^2y^3z^2 - 1.0y^3z^2 - 2.0x^2z^2 + 2.0z^2 + 1.0x^2y^3z^1 - 1.0y^3z^1 \\
& -2.0x^2z^1 + 2.0z^1 - 2.0x^2y^3 + 2.0y^3 + 4.0x^2 - 4321.0
\end{aligned}
\tag{4.9}
$$

with maximum degree $n_1 = 2$ in x, $n_2 = 3$ in y, and maximum degree $n_3 = 2$ in z. The indices given to the right of each row of the matrix are the indices of the coefficient of the polynomial that occurs at the corresponding position of the right hand side. The values of the coefficients not equal to zero are listed in the rigthmost column. Notice, that the matrix is never build up explicitely during program execution.

Matrix structure:

```
1...................................  2 3 2    1.000
x1..................................  1 3 2
.x1.................................  0 3 2   -1.000
...1................................  2 2 2
...x1...............................  1 2 2
..y.x1..............................  0 2 2
......1.............................  2 1 2
......x1............................  1 1 2
.....y.x1...........................  0 1 2
.........1..........................  2 0 2   -2.000
.........x1.........................  1 0 2
........y.x1........................  0 0 2    2.000
............1.......................  2 3 1    1.000
............x1......................  1 3 1
.............x1.....................  0 3 1   -1.000
...............1....................  2 2 1
...............x1...................  1 2 1
..............y.x1..................  0 2 1
..................1.................  2 1 1
..................x1................  1 1 1
.................y.x1...............  0 1 1
.....................1..............  2 0 1   -2.000
.....................x1.............  1 0 1
...........z........y.x1............  0 0 1    2.000
........................1...........  2 3 0   -2.000
........................x1..........  1 3 0
.........................x1.........  0 3 0    2.000
...........................1........  2 2 0
...........................x1.......  1 2 0
..........................y.x1......  0 2 0
..............................1.....  2 1 0
..............................x1....  1 1 0
.............................y.x1...  0 1 0
.................................1..  2 0 0    4.000
.................................x1.  1 0 0
.......................z........y.x1  0 0 0    4321.000
```

In this case one has $dist_1 = 1$, $dist_2 = 3$, $dist_3 = 12$ and $dist_4 = 36 = dim$ (see (4.7)). The characters `x`, `y`, and `z` indicate the positions wehre the values $-x$, $-y$, and $-z$ occur as entries of the sparse matrix. The matrix structure shown corresponds to the polynomial $p(x, y, z)$ of Exercise 1.

4.3 Algorithmic Description

With the notation (see (4.5))

$$A := A^{(r)} \text{ and } b := b^{(r)}$$

the linear system $A \cdot x = b$ corresponding to the polynomial

$$p(t_1, t_2, \cdots, t_r) = \sum_{i_r=0}^{n_r} \cdots \sum_{i_2=0}^{n_2} \sum_{i_1=0}^{n_1} a_{i_1, i_2, \cdots, i_r} t_1^{i_1} t_2^{i_2} \cdots t_r^{i_r}$$

in r variables is to be solved.

A first approximation of the solution of this system is computed by the nested Horner scheme. This is equivalent to a forward substitution step for the lower triangular system. This step may be coded in PASCAL-XSC as follows:

```
FUNCTION initial_approx( b   : PolVector;
                         pnt : ArgVector;
                         Dist: Intvector): rvector[1..Ub(b)];

{ Computation of a first approximation to the sparse system A*x = b }

VAR  i, k: integer;
     x   : rvector[1..Ub(b)];
BEGIN
  FOR i:= 1 TO Ub(b) DO        { Loop over all rows of the sparse system }
  BEGIN
    x[i]:= b[i];               { x[i] is to be computed                   }
   { Loop over all variables xk, k= 1, 2, ...,r: }
    FOR k:= 1 TO Ub(pnt) DO BEGIN
      IF occurrence(i, k, Dist) THEN
        x[i]:= x[ i-dist[k] ] * pnt[k] + x[i];     { typical Horner step }
    END;
  END;
  initial_approx:= x;            { x[dim] is an approximation to p(pnt) }
END;
```

The result of this function is a dynamic array with dim components. The function `occurrence(i,k,dist)` checks whether the variable t_k of the polynomial occurs in the i-th row of the matrix. Formula (4.8) is used for this check. The $r + 1$ components of the vector `dist` are defined according to (4.7).

The approximation is improved step by step using residual iteration [285] until the desired accuracy is reached for the last component of the solution vector, which is the value of the polynomial. The complete algorithm may be summarized as follows:

Algorithm 4.1: MultiDimPolyEval () {procedure}

1. Floating-point computation:
   Compute an approximation $x^{(0)}$ with $A \cdot x^{(0)} \approx b$ using forward substitution
2. Interval iteration:
   $k := 0$
   repeat
   (a) Compute an inclusion of the actual defect:
   $k := k + 1$
   $[d]^{(k)} := \Diamond(b - \sum_{j=0}^{k-1} Ax^{(j)}), \qquad \Diamond:$ rounding to an enclosing interval
   (b) Compute $A[x]^{(k)} = [d]^{(k)}$ (forward substitution to get $[x]^{(k)}$):
   for i:=1 **to** dim **do**
   $$[x]_i^{(k)} := \Diamond([d]_i^{(k)} \; - \; \sum_{j=0}^{i-1} A_{ij} \cdot [x]_j^{(k)})$$
   (c) Enclosure of the last component of $A^{-1} \cdot b$:
   $[y] := \Diamond([x]_{dim}^{(k)} + \sum_{j=1}^{k-1} x_{dim}^{(j)})$
   (d) $x^{(k)} := \text{mid}([x]^{(k)})$ (prepare next iteration step)

   until $(\text{diam}[y]) \leq 1ulp$ **or** $(k \geq k_{max})$
3. The following has been verified: $\quad p(t_1, \ldots, t_r) \in [y]$

The notation $\text{diam}[y] \leq 1ulp$ means that the bounds of the interval $[y]$ are either the same or two adjacent floating-point numbers.

The numerically critical step 2a) is the computation of the defect $[d]^{(k)}$. Therefore it is done using the optimal dot product. The following listing shows the PASCAL-XSC implementation for this step:

```
FUNCTION defect( x   : rmatrix;
                 b   : PolVector;
                 pnt : ArgVector;
                 iter: integer ): ivector[1..Ub(b)];

{ Interval enclosure of the defect with respect to the actual }
{ approximation for the solution of the system               }

VAR  i, k, j    : integer;
     ActDegree  : IntVector[1..Ub(pnt)];
     d          : ivector[1..Ub(b)];
     accu       : dotprecision;

BEGIN
```

```
    { loop over all rows of the sparse system }
    FOR i:= 1 TO Ub(b) DO BEGIN
      {                 0    1          iter   }
      {   d := b - A*( x   + x   + ... + x     )  }
      accu:= #( b[i] - FOR j:= 0 TO iter SUM( x[i,j] ) );
      FOR k:= 1 TO Ub(pnt) DO  { loop over all variables tk }
        IF occurrence(i, k, Dist) THEN { tk appears in actual row }
           accu:= #( accu + FOR j:=0 TO iter SUM(x[i-dist[k], j]*pnt[k]) );
      defect[i]:= ##( accu );
    END;
  END;
```

The result of this function is a dynamic array with *dim* interval components. The following holds:

$$defect = [d]^{(\text{iter}+1)} = \Diamond(b - Ax^{(0)} - Ax^{(1)} - Ax^{(2)} - \cdots - Ax^{(\text{iter})}).$$

The vector `x[*,j]` corresponds to the j-th improvement $x^{(j)}$ of the solution of the sparse system.

4.4 Implementation and Examples

4.4.1 PASCAL-XSC Program Code

```
MODULE polmod;

{***   Accurate Evaluation of Polynomials in Several Variables   ***}


{   r         : number of variables of the polynomial                  }
{   dim       : dimension of the corresponding sparse system           }
{   MaxDegree : maximum degree of each variable                        }
{   pnt       : evaluation of the polynomial is done at pnt            }
{   b         : coefficients of the polynomial                         }
{   ItNum     : number of required iterations                          }
{   PolVal    : enclosure of the polynomial value                      }
{   approx    : value of the polynomial computed with Horner's scheme }

USE i_ari, mv_ari;

GLOBAL TYPE
   PolVector  = DYNAMIC ARRAY[*] OF real;
   IntVector  = DYNAMIC ARRAY[*] OF integer;
   ArgVector  = GLOBAL DYNAMIC ARRAY[*] OF real;

VAR
  Poly: text;     { File with data of the polynomial }
  TEST: boolean;  { Output of intermediate results ? }

OPERATOR <> (a, b: IntVector) ia_ne_ia: boolean; {ia means integer array }
VAR i  : integer;
    neq: boolean;
BEGIN
   neq:= false;
   FOR i:= LB(a) TO UB(a) DO
     neq:= neq OR (a[i] <> b[i]);
   ia_ne_ia:= neq;
```

```
END;


OPERATOR = (a, b: IntVector) ia_eq_ia: boolean;
VAR i : integer;
    eq: boolean;
BEGIN
   eq:= true;
   FOR i:= LB(a) TO UB(a) DO
      eq:= eq AND (a[i] = b[i]);
   ia_eq_ia:= eq;
END;


PROCEDURE decrement( VAR ActDegree: IntVector; MaxDegree: IntVector);

{ The actual constellation of the degrees of the variables of the         }
{ polynomial is decremented by 1 beginning with the leftmost index        }
{ which corresponds to the first variable x1.                             }

{ Examples: 2 3 4 ---> 1 3 4,    0 0 4 ---> MaxDegree[1] MaxDegree[2] 3.  }
{ MaxDegree[k] indicates the maximum degree of the k-th variable xk.      }
{ r indicates the number of variables of the polynomial.                  }

VAR
   k: integer;
   r: integer;

BEGIN
   r:= UB(MaxDegree);
   k:= 1;  { start wirth the leftmost index corresponding to x   }
   REPEAT  {                                                 1   }
      IF ActDegree[k] > 0 THEN BEGIN
         ActDegree[k]:= ActDegree[k] - 1;
         k:= r;                        { force stopping }
      END
      ELSE ActDegree[k]:= MaxDegree[k]; { proceed with the index        }
      k:= k+1;                         { belonging to the variable x    }
   UNTIL ( k > r );                    {                           k+1 }
END;

FUNCTION occurrence( i, k: integer; dist: IntVector): boolean;
BEGIN
   occurrence:= ( (i MOD dist[k] = 0) AND
                  (NOT(i MOD dist[k+1] = dist[k])) );
END;

FUNCTION initial_approx( b   : PolVector;
                         pnt : ArgVector;
                         Dist: Intvector): rvector[1..Ub(b)];

{ Computation of a first approximation to the sparse system A*x = b }

VAR  i, k: integer;
     x   : rvector[1..Ub(b)];
BEGIN
  FOR i:= 1 TO Ub(b) DO        { Loop over all rows of the sparse system }
  BEGIN
    x[i]:= b[i];               { x[i] is to be computed                  }
   { Loop over all variables xk, k= 1, 2, ...,r: }
    FOR k:= 1 TO Ub(pnt) DO BEGIN
      IF occurrence(i, k, Dist) THEN
        x[i]:= x[ i-dist[k] ] * pnt[k] + x[i];     { typical Horner step }
    END;
```

```
  END;
  initial_approx:= x;           { x[dim] is an approximation to p(pnt) }
END;

GLOBAL PROCEDURE StopProgram;
VAR x: real;
BEGIN x:= x/(x-x); END;

GLOBAL FUNCTION ReadYesNoQuit(): char;
VAR c   : char;
    exit: boolean;
BEGIN
   REPEAT
      WHILE input↑ = ' ' DO get(input);
      read(c);
      exit:=  c IN ['j', 'J', 'n', 'N', 'y', 'Y'];
      IF NOT exit THEN
         writeln('Allowed characters are: j, J, n, N, y, Y, q, Q');
      IF c IN ['q', 'Q'] THEN StopProgram;
   UNTIL exit;
   ReadYesNoQuit:= c;
END;

PROCEDURE SkipComment(VAR infile: text);
BEGIN
  REPEAT
    WHILE (infile↑ = ' ') AND (NOT eoln(infile)) DO BEGIN
      get(infile);
    END;
    IF eoln(infile) OR (infile↑ = '$') THEN readln(infile);
    { The symbol $ is used to indicate comment lines }
  UNTIL ( NOT(eoln(infile)) AND NOT (infile↑ IN['$', ' ']) );
END;

GLOBAL PROCEDURE Initialize(VAR NumberOfVariables   : integer;
                            VAR NumberOfCoefficients: integer;
                            VAR Poly                : text);
VAR k, dim, degree: integer;
BEGIN
   reset(Poly);
   SkipComment(Poly);
   read(Poly, NumberOfVariables);
   IF TEST THEN writeln('Number of variables: ', NumberOfVariables);
   dim:= 1;
   FOR k:= 1 TO NumberOfVariables DO BEGIN
      SkipComment(Poly);
      read(Poly, degree); { Maximal Degree of variable x[k] }
      dim:= dim*(degree+1);
   END;
   IF TEST THEN writeln('dim: ', dim);
   NumberOfCoefficients:= dim;
END;

GLOBAL PROCEDURE initialize_dynamic_arrays
                                  ( VAR p         : PolVector;
                                    VAR MaxDegree : IntVector;
                                    VAR Dist      : IntVector;
                                    VAR Poly      : text );
VAR
   ActDegree    : IntVector[1..UB(MaxDegree)];
   h_exp        : IntVector[1..UB(MaxDegree)];
   i, k         : integer;
   r, dim       : integer;
   IntegerDummy : integer;
```

```
    coeff        : real;
    exit         : boolean;
  BEGIN
    r:=   Ub(MaxDegree);
    dim:= Ub(p);
    reset(Poly);
    SkipComment(Poly);
    read(Poly, IntegerDummy); { NumberOfVariables }
    FOR k:= 1 TO r DO BEGIN
       SkipComment(Poly); read(Poly, MaxDegree[k]);
    END;

    { dist[k] gives the distance of the k-th variable }
    {   from the main diagonal                        }
    dist[1]:= 1;
    FOR k:= 1 TO r DO dist[k+1]:= dist[k]*(MaxDegree[k] + 1);
    FOR i:= 1 TO dim DO p[i]:= 0; { initialize all coeff. with 0 }
    IF TEST THEN BEGIN
       writeln; writeln;
       writeln('Input of the coefficients of the polynomial ...');
       writeln;
    END;
    exit:= false;
    REPEAT
       { writeln;                                                 }
       { write('Coefficient and combination of actual degrees?  '); }
       SkipComment(Poly);
       read( Poly, coeff );
       IF TEST THEN writeln('coeff:', coeff);
       FOR i:= 1 TO r DO BEGIN
         READ(Poly, k);
         exit:= NOT ( (0 <= k) AND (k<= MaxDegree[i]) ) OR exit;
         ActDegree[i]:= k;
       END;
       readln(Poly);
       IF ( NOT exit ) THEN BEGIN
          h_exp:= MaxDegree;
          i:= 1;
          WHILE ( h_exp <> ActDegree ) DO BEGIN
             decrement( h_exp, MaxDegree );
             i:= i+1;
          END;
          p[i]:= coeff;
          IF TEST THEN writeln('p[',i,']=', p[i]);
       END
    UNTIL exit;
  END;

  GLOBAL PROCEDURE ReadPoint(VAR pnt: ArgVector);
  VAR k: integer;
  BEGIN
    FOR k:= 1 TO Ub(pnt) DO BEGIN
      write('        t[',k:0,'] =   ?   ');
      read(pnt[k]);
    END;
  END;

  GLOBAL PROCEDURE WritePoint(pnt: ArgVector);
  VAR k: integer;
  BEGIN
    writeln('Argument:  ');
    FOR k:= 1 TO Ub(pnt) DO BEGIN
      writeln('   t[',k:0,'] = ', pnt[k]);
    END;
```

```
  writeln;
END;

GLOBAL FUNCTION Horner(b         : PolVector;
                       pnt       : ArgVector;
                       MaxDegree : IntVector;
                       Dist      : IntVector): real;
{ Horner's scheme exploiting the sparse matrix structure         }
VAR z: RVector[1..UB(b)];                { local to this function }
BEGIN
  z:= Initial_approx( b, pnt, Dist );
  Horner:= z[UB(z)];  { z[dim] is an approximation to p(pnt) }
END;

GLOBAL FUNCTION EvalPol(p         : PolVector;
                        pnt       : ArgVector;
                        MaxDegree : IntVector;
                        Dist      : IntVector): interval;

CONST MaxIt = 15;   { maximum number of iterations allowed }
VAR
   ItNum  : integer;
   approx : real;
   PolVal : interval;

 FUNCTION defect( x   : rmatrix;
                  b   : PolVector;
                  pnt : ArgVector;
                  iter: integer ): ivector[1..Ub(b)];

 { Interval enclosure of the defect with respect to the actual }
 { approximation for the solution of the system                }

 VAR  i, k, j   : integer;
      ActDegree : IntVector[1..Ub(pnt)];
      d         : ivector[1..Ub(b)];
      accu      : dotprecision;

 BEGIN
   { loop over all rows of the sparse system }
   FOR i:= 1 TO Ub(b) DO BEGIN
     {                 0    1          iter   }
     {   d := b - A*( x   + x   + ... + x    )  }
     accu:= #( b[i] - FOR j:= 0 TO iter SUM( x[i,j] ) );
     FOR k:= 1 TO Ub(pnt) DO   { loop over all variables tk }
       IF occurrence(i, k, Dist) THEN { tk appears in actual row }
          accu:= #( accu + FOR j:=0 TO iter SUM(x[i-dist[k], j]*pnt[k]) );
     defect[i]:= ##( accu );
   END;
 END;

 FUNCTION solve( pnt: ArgVector; d: ivector): ivector[1..Ub(d)];

 { computation of A*[x1] = [d] using forward substitution }

 VAR  i, k: integer;
      erg : ivector[1..Ub(d)];
 BEGIN
   erg:= d;
   FOR i:= 1 TO Ub(d) DO
   BEGIN
     FOR k:= 1 TO Ub(pnt) DO
     BEGIN
       IF occurrence(i, k, Dist) THEN
```

```
        BEGIN
          erg[i]:= erg[i-dist[k]]*pnt[k] + erg[i];
        END;
      END; { loop over all variables tk }
    END; { rows }
    solve:= erg;
  END; { function solve }

  PROCEDURE pol_value( pnt: ArgVector; b: PolVector;
                       VAR iter: integer;
                       VAR pol_val: interval; VAR approx: real );

  VAR  x     : rmatrix[1..Ub(b), 0..MaxIt];
       x1, d: ivector[1..Ub(b)];
       i     : integer;

  BEGIN
    iter:= 0;      { it : number of iterations so far          }
    x[*,0]:= initial_approx( b, pnt, Dist );
    approx:= x[Ub(b),0];   { value of the polynomial using Horner's scheme }

    REPEAT
      { improved approximation =                        }
      {    approximation + midpoint of the enclosure     }
      {    of the solution of the system with the actual }
      {    defect as right hand side                     }
      IF iter>0 THEN
        FOR i:=1 TO Ub(b) DO x[i,iter]:= MID( x1[i] );
      d:= defect( x, b, pnt, iter ); { compute an inclusion of the defect }
      x1:= solve( pnt, d ); { compute an inclusion of the interval system }
                               {   with the actual defect d as      }
                               {   right hand side                  }
      pol_val := ##(FOR i:=0 TO iter SUM(x[Ub(b),i])) + x1[Ub(b)];
      { pol_val is an enclosure of the value of the polynomial }
      iter:= succ(iter); { increment the number of iterations done so far }
    UNTIL ( succ(succ(INF(pol_val))) >= SUP(pol_val) ) OR (iter = MaxIt);
  END;

BEGIN { Body of function EvalPol }
   pol_value(pnt, p, ItNum, PolVal, approx);
   IF TEST THEN BEGIN
     writeln('Number of iterations: ', ItNum);
     writeln('First approximation: ', approx);
  END;
  EvalPol:= PolVal;
END; { of EvalPol }

GLOBAL PROCEDURE PolWrite( b: PolVector; MaxDegree, Dist: IntVector );
{ displays the axctual polynomial on the screen }
VAR actual_degree: IntVector[1..Ub(MaxDegree)];
     i, j, num   : integer;
     r: integer;
BEGIN
  r:= Ub(MaxDegree);
  writeln; writeln;
  writeln( 'Evaluation of the polynomial: ' );
  writeln;
  IF r=1 THEN write( '  P(x) =     ' );
  IF r=2 THEN write( '  P(x,y) =   ' );
  IF r=3 THEN write( '  P(x,y,z) = ' );
  IF r>3 THEN {  write( '  P  =       ' ); }
  BEGIN
    write( '  P(t1,t2,t3' );
    FOR j:= 4 TO r DO write( ',t', j:1 );
```

```
    write( ') = ' );
    writeln; writeln;
    write( '               ' );
  END;

  num:= 0;
  actual_degree:= MaxDegree;

  FOR i:= 1 TO Ub(b) DO
  BEGIN
    IF b[i] <> 0 THEN
    BEGIN
      IF b[i] > 0 THEN write(' + ') ELSE write(' - ');
      num:= num + 1;
      write( abs(b[i]):3:1 );
      IF r <= 3 THEN
      BEGIN
        IF (actual_degree[1] <> 0) THEN write( ' x↑', actual_degree[1]:1 );
        IF r > 1 THEN IF (actual_degree[2] <> 0) THEN
          write( ' y↑', actual_degree[2]:1 );
        IF r > 2 THEN
          IF (actual_degree[3]<>0 ) THEN write(' z↑', actual_degree[3]:1);
      END
      ELSE
        FOR j:=1 TO r DO
          IF actual_degree[j]<>0 THEN
            write(' t', j:1,'↑', actual_degree[j]:1);
      IF num MOD 3 = 0 THEN
      BEGIN
        writeln; writeln;
        write( '               ' );
      END;
    END;
    decrement(actual_degree, MaxDegree);
  END;
  writeln; writeln;
END; { of PolWrite }

GLOBAL PROCEDURE MatrixWrite( b: PolVector; MaxDegree, Dist: IntVector );

{ Displays the structure of the sparse matrix associated with the      }
{ polynomial in use.                                                   }
{ x, y, z and * indicate nonzero matrix elements, "." corresponds to 0 }

VAR i, k, j      : integer;
    r, dim       : integer;
    ActualDegree : IntVector[1..UB(MaxDegree)]; { actual combination   }
                                                {           of degrees }
    row          : DYNAMIC ARRAY[1..UB(B)] OF char;

BEGIN
  r:= UB(MaxDegree);
  dim:= UB(b);
  ActualDegree:= MaxDegree;
  i:= 1;

  REPEAT { check occurrence of k-th (k=1...r) variable in row i }
    k:= 1;
    FOR j:= 1 TO dim DO row[j]:= '.';
    row[i]:='1';

    REPEAT
      IF ActualDegree[k] < MaxDegree[k] THEN
      BEGIN
```

```
      CASE k OF
        1    : row[i-dist[k]]:= 'x';   { first variable occurres }
        2    : row[i-dist[k]]:= 'y';   { second    "      "      }
        3    : row[i-dist[k]]:= 'z';   { third     "      "      }
      ELSE   : row[i-dist[k]]:= '*';   { k-th (k>3)"      "      }
      END;
      IF ActualDegree[k] <> 0 THEN k:= r; { force stopping }
    END
    ELSE k:= r;                          { force stopping }
    k:= k+1;              { proceed to the next variable }
  UNTIL ( k > r );

  FOR j:=1 TO dim DO write( row[j] );      { display row i }
  write (' ');
  FOR k:= 1 TO r DO write( ActualDegree[k]:2);{ display constellation }
  {writeln; }                       { of degrees of right hand side }
  IF ( NOT ( b[i] = 0 ) ) THEN writeln( ' ', b[i] ) ELSE writeln;

  decrement(ActualDegree, MaxDegree);
  i:= i+1;          { proceed to the next row }
  IF (i MOD 23 = 0) AND (i <= dim) THEN
  BEGIN  { new page }
    write( '  Press  <ENTER>  to continue ' );
    readln;
  END;
 UNTIL ( i > dim );
 write( '  Press  <ENTER>  to continue ' );
 readln;
END; { of MatrixWrite }

{**********************************************************************}
{   Initialization part                                                }
{**********************************************************************}
BEGIN { Body of module polmod }
   TEST:= false; { Suppress output of intermediate results }
END.  { of polmod             }


PROGRAM PolMain;

USE PolMod, mv_ari, i_ari;

PROCEDURE main(
       NumberOfVariables, NumberOfCoefficients: integer; VAR Poly: text);
VAR
   t         : ArgVector[1..NumberOfVariables];
   MaxDegree : IntVector[1..NumberOfVariables];
   p         : PolVector[1..NumberOfCoefficients];
   Dist      : IntVector[1..NumberOfVariables+1];
   HornerValue  : real;
   AccurateValue: interval;
BEGIN
   initialize_dynamic_arrays(p, MaxDegree, Dist, Poly);
   PolWrite(p, MaxDegree, Dist);
   write('Output of sparse matrix structure ?  ');
   IF ReadYesNoQuit IN ['y', 'Y', 'j', 'J']  THEN BEGIN
      writeln; MatrixWrite(p, MaxDegree, Dist);
   END;
   REPEAT { do forever }
      writeln;
      writeln('Argument t = ?  (or produce an error to quit: q)');
      ReadPoint(t);
      WritePoint(t);
      HornerValue  := Horner(  p, t, MaxDegree, Dist );
```

```
      AccurateValue:= EvalPol( p, t, MaxDegree, Dist );
      writeln('Horner:');
      writeln('  ', HornerValue);
      writeln('Correct enclosure:');
      writeln(AccurateValue);
   UNTIL true=false;
END;

VAR Poly: text; { Data file with specification of the polynomial }
    NumberOfVariables, NumberOfCoefficients: integer;
BEGIN
   { Reset the data file which holds the data of the polynomial: }
   reset(Poly, 'polynom.dat');
   { Read the number of variables and compute the             }
   {   number of coefficients using the maximal degrees:      }
   Initialize(NumberOfVariables, NumberOfCoefficients, Poly);
   { Create all dynamic arrays associated with the polynomial }
   {   evaluation:                                            }
   Main(NumberOfVariables, NumberOfCoefficients, Poly);
END.
```

4.4.2 Numerical Examples

First the polynomial in two variables

$$p(x,y) = -1.0y^6 + 3.0x^1y^5 - 5.0x^3y^3 + 3.0x^5y^1 + 1.0x^6$$

is considered. The sparse matrix structure corresponding to this polynomial is shown and the polynomial is evaluated at the points (4181,6765), (17711,28657) and (3,7).

```
Evaluation of the polynomial:

  P(x,y) =    - 1.0 y^6 + 3.0 x^1 y^5 - 5.0 x^3 y^3

               + 3.0 x^5 y^1 + 1.0 x^6

Output of sparse matrix structure ?  YES
1..............................................  6 6
x1.............................................  5 6
.x1............................................  4 6
..x1...........................................  3 6
...x1..........................................  2 6
....x1.........................................  1 6
.....x1........................................  0 6  -1.000000000000000E+000
.......1.......................................  6 5
.......x1......................................  5 5
........x1.....................................  4 5
.........x1....................................  3 5
..........x1...................................  2 5
...........x1..................................  1 5   3.000000000000000E+000
......y.....x1.................................  0 5
..............1................................  6 4
..............x1...............................  5 4
...............x1..............................  4 4
................x1.............................  3 4
.................x1............................  2 4
..................x1...........................  1 4
............y.....x1...........................  0 4
.....................1.........................  6 3
.....................x1........................  5 3
```

```
......................x1.........................  4 3
.......................x1........................  3 3 -5.000000000000000E+000
........................x1.......................  2 3
.........................x1......................  1 3
....................y.....x1.....................  0 3
............................1....................  6 2
............................x1...................  5 2
.............................x1..................  4 2
..............................x1.................  3 2
...............................x1................  2 2
................................x1...............  1 2
...........................y.....x1..............  0 2
...................................1.............  6 1
...................................x1............  5 1  3.000000000000000E+000
....................................x1...........  4 1
.....................................x1..........  3 1
......................................x1.........  2 1
.......................................x1........  1 1
..................................y.....x1.......  0 1
..........................................1......  6 0  1.000000000000000E+000
..........................................x1.....  5 0
...........................................x1....  4 0
............................................x1...  3 0
.............................................x1..  2 0
..............................................x1.  1 0
.........................................y.....x1  0 0
  Press  <ENTER>  to continue

Argument x = ?
        x[1] =   ?         x[2] =   ?   Argument:
   x[1] =  4.181000000000000E+003
   x[2] =  6.765000000000000E+003

Horner:
   1.048576000000000E+006
Correct enclosure:
[  1.000000000000000E+000,  1.000000000000000E+000 ]

Argument x = ?
        x[1] =   ?           x[2] =   ?   Argument:
   x[1] =  1.771100000000000E+004
   x[2] =  2.865700000000000E+004

Horner:
  -1.288490188800000E+010
Correct enclosure:
[ -1.000000000000000E+000, -1.000000000000000E+000 ]

Argument x = ?
        x[1] =   ?           x[2] =   ?   Argument:
   x[1] =  3.000000000000000E+000
   x[2] =  7.000000000000000E+000

Horner:
  -6.859000000000000E+003
Correct enclosure:
[ -6.859000000000000E+003, -6.859000000000000E+003 ]
```

The second example uses the polynomial (4.9) the matrix structure of which is shown at the end of Paragraph 4.2:

```
Evaluation of the polynomial:

  p(x,y,z) =  + 1.0 x^2 y^3 z^2 - 1.0 y^3 z^2 - 2.0 x^2 z^2

               + 2.0 z^2 + 1.0 x^2 y^3 z^1 - 1.0 y^3 z^1

               - 2.0 x^2 z^1 + 2.0 z^1 - 2.0 x^2 y^3

               + 2.0 y^3 + 4.0 x^2  + 4321

(M)atrix structure / (P)oint / (Q)uit?      M
```

4.4.3 Input Procedures Using Data Files

A comfortable specification of a multidimensional polynomial is provided. In the first line of the data file the number of variables is given. In the second line the maximum degree of each variable is specified. The number of integer values given there correspondes to the number of variables of the polynomial. In the subsequent lines the coefficients not equal to zero are listed together with their corresponding degrees of the variables of the polynomial. The end of the data file is indicated by a constellation of actual degrees which is not valid (for example: negative degrees). Comments in the data file are skipped automatically. A typical input file specifying polynomial in several variables is shown. The polynomial specified here is the polynomial (4.1).

```
$---------------------------------------------------------------
$ Notice: The dollar symbol $ indicates a comment for the rest
$    of the actual line
$---------------------------------------------------------------
$    This file may be used in program polmain as input file
$
$   When reading this file, comment is skipped automatically
$---------------------------------------------------------------
$ Data for a polynomial in several variables:
$--------------------------------------------
 2      $ number of variables
 6  6   $ maximum degrees of the variables

$ The data represent a fibonacci polynomial in 2 variables

 1      6  0                 $  1*t[1]^6*t[2]^0 =  t[1]^6
 3      5  1                 $  3*t[1]^5*t[2]^1
-5      3  3                 $ -5*t[1]^3*t[2]^3
 3      1  5                 $  3*t[1]^1*t[2]^5
-1      0  6                 $ -1*t[1]^0*t[2]*6  = -t[2]^6
 0  -1 -1    $ negative degree! ==>        end of data
             $   all remaining coefficients are zero
$---------------------------------------------------------------
```

The second input file specifies the polynomial (4.9) at the end of Paragraph 4.2:

```
$---------------------------------------------------------------
$    This file may be used in program polmain as input file
$
$   When reading this file, comment is skipped automatically
$---------------------------------------------------------------
$ Data for a polynomial in several variables:
```

```
$--------------------------------------------
 3         $ number of variables
 2  3  2   $ maximum degrees of the variables

 $ The data represent a polynomial p in 3 variables
 $ It holds p(x, y, -2) =  4325 for arbitrary x and y

 1      2  3  2         $    1*t[1]^2*t[2]^3*t[3]^2
 1      2  3  1         $    1*t[1]^2*t[2]^3*t[3]^1
-2      2  3  0         $   -2*t[1]^2*t[2]^3*t[3]^0
-1      0  3  2         $   -1*t[1]^0*t[2]^3*t[3]^2
-1      0  3  1         $   -1*t[1]^0*t[2]^3*t[3]^1
 2      0  3  0         $    2*t[1]^0*t[2]^3*t[3]^0
-2      2  0  2         $   -2*t[1]^2*t[2]^0*t[3]^2
-2      2  0  1         $   -2*t[1]^2*t[2]^0*t[3]^1
 4      2  0  0         $    4*t[1]^2*t[2]^0*t[3]^0
 2      0  0  2         $    2*t[1]^0*t[2]^0*t[3]^2 = 2*t[3]^2
 2      0  0  1         $    2*t[1]^0*t[2]^0*t[3]^1 = 2*t[3]
 4321   0  0  0         $ 4321*t[1]^0*t[2]^0*t[3]^0 = 4321
 0     -1 -1 -1   $ negative degree! ==>       end of data
                  $    all remaining coefficients are zero
$-----------------------------------------------------------
```

4.5 Restrictions and Hints

The final algorithm takes advantage of the very special form of the system matrix. In fact, this lower triangular sparse matrix is never built up explicitly in the memory of the computer. An optimal scalar product [214] is essential to make the sparse system algorithm work numerically. To get verified bounds, interval arithmetic [13] is used. Programming languages like FORTRAN-SC[1] [141], PASCAL-XSC or C-XSC [175] offer these features and are therefore essential tools for an implementation of such an algorithm.

4.6 Exercises

Exercise 1:

Let us consider the following polynomial in three variables:

$$\begin{aligned} p(x,y,z) = x^2y^3z^2 \quad &+\quad x^2y^3z - 2x^2y^3 - y^3z^2 - y^3z \\ &+\quad 2y^3 - 2x^2z^2 - 2x^2z + 4x^2 + 2z^2 + 2z + 4321 \end{aligned}$$

The data file for this polynomial is shown in Paragraph 4.4.3.
For arbitrary values x and y

$$p(x, y, -2) = 4325$$

holds.

[1]FORTRAN-SC is now available as the IBM program product ACRITH-XSC (High Accuracy Arithmetic-Extended Scientific Computation, program number 5684-129).

Show that Horner's scheme (as implemented in the given module `polmod`) leads to

$$\begin{aligned} p(1234, 5678, -2) &\approx 4352 \\ p(3333, 5555, -2) &\approx 4608 \\ p(7777, 7777, -2) &\approx 8192 \\ p(12345, 6789, -2) &\approx 0 \end{aligned}$$

whereas the new algorithm always computes the exact value 4321.

Exercise 2:
Use Horner's scheme to compute approximations for the polynomial

$$q(x) = x^4 - 8x^3 + 24x^2 - 32x + 16 = (x-2)^4$$

for the arguments $x_1 := \mathrm{pred}(\square(1.99991))$ and $x_2 := \mathrm{succ}(\square(1.99991))$. Here $\square$ means rounding to the nearest IEEE double number. The functions pred and succ return the next floating point number which is smaller or larger than the actual argument. Show that for the first argument x_1 the magnitude of the approximation is wrong. For the second argument x_2 even the sign of the approximation is wrong. Compute the correct values using the sparse system algorithm. Compute also $q(\square 1.99991)$. How sensitive are the correct results with respect to different rounding modes for x = 1.99991?

4.7 References and Further Readings

A very similar algorithm can be derived for polynomials in several variables with complex coefficients [198]. An algorithm for computing sharp enclosures of polynomials with interval coefficients can be found in [102]. For more information about the computation of sharp enclosures of the range of interval polynomials consult the literature cited in [102].

Chapter 5

Automatic Differentiation

The computation of derivatives is a task which one encounters in numerical analysis very often. Whenever a Newton method is to be applied then the derivative or the Jacobian matrix of the functions involved in the problem must be computed somehow. Also optimization methods often use gradients or even Hesse matrices.

Many numerical textbooks consider it as difficult to compute such derivatives. Numerical libraries on computers usually put the burdon of supplying subroutines for derivatives to the user of these libraries. Alternatively, derivatives are often replaced by finite difference quotients which brings into play many unnecessary additional problems like truncation errors and cancellation errors. Because of these apparent difficulties many numerical textbooks do not even try to consider methods which need derivatives of even higher order, like methods based on Taylor expansions.

Sometimes the use of symbolic differentiation is recommended to produce expressions for the needed derivatives. Almost all computer algebra systems can be utilized for this purpose. Many of them have features which allow the derived expressions to be converted to the syntax of some programming language. However, the expressions for the derivatives obtained in this way are often extremely complex and therefore costly to evaluate and sometimes they may even be numerically unstable to evaluate.

For many years already there exist interesting and very efficient methods to compute derivative values *exactly* (if exact arithmetic is used) for a large class of functions, i.e. in principle all functions which can be programmed in a common programming language. These methods are known as *automatic differentiation* and were developed already since the 60's by e.g. Moore, [247], and later by Rall, [267] – [272], Griewank, [114], [115], Iri, [148], Corliss, [86], [87], Fischer, [101] – [104] and many others. Automatic differentiation is only now starting to be recognized as a very useful tool by a wider audience in the community of numerical analysts. An extensive bibliography compiled by G. Corliss is contained in [115].

The main difference between these methods and finite differences and symbolic differentiation is that automatic differentiation computes derivative *values* which are the *exact* values of the derivatives (if exact arithmetic is used in the computation). I.e. automatic differentiation is a numerical method just as finite differences but in contrast to the latter it produces *exact* results. On the other side symbolic differentiation also produces exact results, however, there *expressions* for the derivatives are calculated. Numerical *values* of the derivatives are then obtained by evaluating the expressions numerically for suitable parameters. This indirect way makes symbolic differentiation often much more costly. Automatic differentiation can be viewed as a combination of symbolic differentiation and numerical evaluation performed simultaneously. I.e. the differentiation rules are applied to the given function but they are applied immediately to the numerical values not first to expressions. This is the reason why automatic differentiation is exact just as symbolic differentiation but at the same time more efficient.

Also, during the last years many variants of automatic differentiation have been developed which differ primarily in the order in which the differentiation rules are applied. We can mainly distinguish between two different types of methods which are called the *forward mode* and the *reverse mode* of automatic differentiation.

Historically the forward mode was used first whereas the reverse mode was developed later. For functions in several variables the reverse mode is usually very much faster than the forward mode, the difference being very drastic: for a function $f : \mathbb{R}^n \to \mathbb{R}$ the time for computing the gradient using the forward mode is proportional to n times the time required for an evaluation of f. In contrast, for the reverse mode the time to compute the gradient is less than about 5 times the cost for an evaluation of f, see e.g. [114], [148], [101], [104]. Amazingly, this time ratio is *independent* of the dimension n !

In Section 5.1 we will first discuss the one-dimensional case with higher derivatives, i.e. we present an automatic differentiation method for the computation of Taylor coefficients. This method is a forward method. In Section 5.2 we turn to the multidimensional case and discuss the implementation of the reverse mode for the computation of gradients. Finally, in Section 5.3 we discuss and implement a method for the fast computation of directional derivatives. This is a forward method again but it is as efficient as a reverse method.

5.1 Functions in One Variable, Taylor Coefficients

5.1.1 Introduction

The computation of high derivatives of a function is often considered to be a tedious and error-prone task which is the reason why there exist almost no numerical methods making use of high derivatives. Only the Taylor series method for the approximate solution of ordinary differential equations is sometimes mentioned in textbooks. However, this method is usually discarded immediately because it is considered to be far too expensive. Higher derivatives often occur also in remainder terms of various numerical approximation methods such as Taylor expansion, interpolation, numerical integration and the like.

In this section we show that the computation of such higher derivatives is indeed very easy and efficient and that even unknown intermediate values which often appear in remainder terms can be handled almost trivially by the use of interval arithmetic. Here the combination of automatic differentiation and interval arithmetic has a strong effect of synergy: Computing high derivatives efficiently and enclosing unknown intermediate points in intervals results in methods of a totally new quality. This combination makes it possible to estimate remainder terms rigorously and thus to derive proofs automatically on a computer which were not possible earlier (or which were at least much more complicated).

5.1.2 Theoretical Background

To keep the presentation simple assume that $f : I\!R \to I\!R$ is an analytic function (we could as well require f to be only sufficiently smooth). Then $f(t)$ can be expanded in a Taylor series around a point t_0

$$f(t) = \sum_{k=0}^{\infty} \frac{f^{(k)}(t_0)}{k!}(t - t_0)^k \ . \tag{5.1}$$

We introduce the following short notation for the Taylor coefficients $\dfrac{f^{(k)}(t_0)}{k!}$ of f at the point of expansion $t = t_0$

$$(f)_k := \frac{f^{(k)}(t_0)}{k!} = \frac{1}{k!}\frac{d^k f(t_0)}{dt^k} \tag{5.2}$$

or

$$(f(t_0))_k := \frac{f^{(k)}(t_0)}{k!} = \frac{1}{k!}\frac{d^k f(t_0)}{dt^k} \tag{5.3}$$

if we want to focus on the point of expansion t_0. Using (5.2) we can rewrite (5.1) as

$$f(t) = \sum_{k=0}^{\infty}(f)_k(t - t_0)^k \ . \tag{5.4}$$

Now let us consider functions f which are the composition of arithmetic operations and elementary functions. For functions of such a structure we can recursively determine the Taylor coefficients by applying appropriate rules for each single operation occurring in the expression for f.

Starting with the two trivial cases we state that for a constant function $f = c$

$$(c)_0 = c, \quad (c)_k = 0, \quad \text{for } k \geq 1 \tag{5.5}$$

and for the independent variable $f = t$

$$(t)_0 = t_0, \quad (t)_1 = 1, \quad (t)_k = 0, \quad \text{for } k \geq 2 \ . \tag{5.6}$$

From the rules for operating with power series we immediately see that for two functions $u(t)$ and $v(t)$ we have

$$\begin{aligned}
(u \pm v)_k &= (u)_k \pm (v)_k \\
(u \cdot v)_k &= \sum_{j=0}^{k}(u)_j(v)_{k-j} \left(= \sum_{j=0}^{k}(u)_{k-j}(v)_j\right) \quad \text{(Cauchy product)} \\
(u/v)_k &= \frac{1}{(v)_0}((u)_k - \sum_{j=1}^{k}(v)_j(u/v)_{k-j}) \ .
\end{aligned} \tag{5.7}$$

Here the rule for the quotient can easily be obtained as follows: write $w = u/v$, then $u = v \cdot w$ and the product rule from (5.7) gives

$$(u)_k = \sum_{j=0}^{k}(v)_j(w)_{k-j} = (v)_0(w)_k + \sum_{j=1}^{k}(v)_j(w)_{k-j}$$

which must be solved for $(w)_k = (u/v)_k$ to obtain the quotient rule in (5.7).

Because of the two different versions of the product rule in (5.7) many of the formulas in the rest of this section can also be represented in several different ways.

With these rules (5.5) – (5.7) we are already in the position to compute any Taylor coefficient of any rational function in t. We just have to apply these rules recursively for each operation in f.

As for division we can derive a formula for the square root $w = \sqrt{u}$ by writing $w \cdot w = u$ and applying the product rule which gives, for $k \geq 1$,

$$\begin{aligned}
(\sqrt{u})_k &= \frac{1}{2(\sqrt{u})_0}((u)_k - \sum_{j=1}^{k-1}(\sqrt{u})_j(\sqrt{u})_{k-j}) \\
&= \begin{cases} \dfrac{1}{2(\sqrt{u})_0}((u)_k - 2\displaystyle\sum_{j=1}^{(k-1)/2}(\sqrt{u})_j(\sqrt{u})_{k-j}), & \text{if } k \text{ odd} \\ \dfrac{1}{2(\sqrt{u})_0}((u)_k - 2\displaystyle\sum_{j=1}^{(k-2)/2}(\sqrt{u})_j(\sqrt{u})_{k-j} - (\sqrt{u})_{k/2}^2), & \text{if } k \text{ even} \end{cases}
\end{aligned} \tag{5.8}$$

where the last form uses only about half as many operations as the form in the first line. Of course, $(\sqrt{u})_0 = \sqrt{(u)_0}$.

Also for the square function $w = u^2$ we can write down a more economical form than just that for multiplication:

$$(u^2)_k = \sum_{j=0}^{k}(u)_j(u)_{k-j} = \begin{cases} 2\displaystyle\sum_{j=0}^{(k-1)/2}(u)_j(u)_{k-j}, & \text{if } k \text{ odd} \\ 2\displaystyle\sum_{j=0}^{(k-2)/2}(u)_j(u)_{k-j} + (u)_{k/2}^2, & \text{if } k \text{ even .} \end{cases} \tag{5.9}$$

To find similar recursion formulas also for elementary functions, we need a further relationship between the Taylor coefficients of f and its derivative f'. Differentiating (5.4) with respect to t we get a series for f':

$$f'(t) = \sum_{k=1}^{\infty} k(f)_k(t-t_0)^{k-1} = \sum_{k=0}^{\infty}(k+1)(f)_{k+1}(t-t_0)^k \ . \tag{5.10}$$

On the other hand, since $(f')_k = \frac{1}{k!}(f')^{(k)}(t_0)$ we have another series representation of f' :

$$f'(t) = \sum_{k=0}^{\infty}(f')_k(t-t_0)^k \ . \tag{5.11}$$

Equating coefficients of these two representations yields

$$(f')_k = (k+1)(f)_{k+1} \quad \text{or} \quad (f)_{k+1} = \frac{1}{k+1}(f')_k \ . \tag{5.12}$$

This relationsship (5.12) is very important in deriving new recursion formulas for elementary functions and also for other applications like the Taylor series solution of ordinary differential equations.

E.g. to get a formula for the Taylor coefficients of $w = e^{u(t)}$ we differentiate w once with respect to t giving $w' = w \cdot u'$. An application of the product rule from (5.7) yields for $k \geq 1$

$$k(w)_k = (w')_{k-1} = (w \cdot u')_{k-1} = \sum_{j=0}^{k-1}(w)_j(u')_{k-1-j} = \sum_{j=0}^{k-1}(w)_j(k-j)(u)_{k-j}$$

and solving this for $(w)_k = (e^u)_k$ we get:

$$(e^u)_k = \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(e^u)_j(u)_{k-j}, \quad k \geq 1 \tag{5.13}$$

and $(e^u)_0 = e^{(u)_0}$ for $k = 0$ since for any function f we always have trivially

$$(f(u))_0 = f((u)_0) \ .$$

The recursion formulas in the rest of this section therefore are all valid for $k \geq 1$.

In a very similar way we can obtain formulas for the Taylor coefficients of $\sin u$ and $\cos u$:

$$\begin{aligned} (\sin u)_k &= \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\cos u)_j(u)_{k-j} \\ (\cos u)_k &= -\frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\sin u)_j(u)_{k-j} \end{aligned} \tag{5.14}$$

and also for $\sinh u$ and $\cosh u$:

$$\begin{aligned} (\sinh u)_k &= \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\cosh u)_j(u)_{k-j} \\ (\cosh u)_k &= \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\sinh u)_j(u)_{k-j} \ . \end{aligned} \tag{5.15}$$

Obviously these formulas have to be applied pair-wise.

For the power function $w = u^a$ with a constant exponent a we can use the same technique, i.e. from $w' = au^{a-1}u' = a\frac{u^a}{u}u'$ we get $w'u = awu'$ where again the product rule gives

$$(u^a)_k = \frac{1}{k\,(u)_0}\sum_{j=0}^{k-1}(a\,(k-j)-j)(u^a)_j(u)_{k-j} \tag{5.16}$$

Now, the recursion formulas of almost all the other elementary functions which we will include in our PASCAL-XSC program can be derived in the following way: assume $f = f(u(t))$ has a derivative of the form $f' = f'(u)u'$ with $f' = 1/g(u)$, i.e. $f' = u'/g$ where $g = g(u)$ is a function which can also be expressed in a form that enables us to compute its Taylor coefficients. Then $u' = f' \cdot g$ and

$$k(u)_k = (u')_{k-1} = (f'g)_{k-1} = \sum_{j=0}^{k-1}(f')_j(g)_{k-j-1} = \sum_{j=1}^{k} j(f)_j(g)_{k-j} \ . \tag{5.17}$$

Solving for $(f)_k$ yields

$$(f)_k = \frac{1}{(g)_0}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(f)_j(g)_{k-j}) \ . \tag{5.18}$$

Using this relation the Taylor coefficients of the remaining functions can be listed in the following table:

f	g	$(f)_k$
$\ln u$	u	$\frac{1}{(u)_0}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\ln u)_j(u)_{k-j})$
$\tan u$	$\cos^2 u$	$\frac{1}{\cos^2(u)_0}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\tan u)_j(\cos^2 u)_{k-j})$
$\cot u$	$-\sin^2 u$	$\frac{-1}{\sin^2(u)_0}((u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j(\cot u)_j(\sin^2 u)_{k-j})$
$\arcsin u$	$\sqrt{1-u^2}$	$\frac{1}{\sqrt{1-(u)_0^2}}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\arcsin u)_j(\sqrt{1-u^2})_{k-j})$
$\arccos u$	$-\sqrt{1-u^2}$	$\frac{-1}{\sqrt{1-(u)_0^2}}((u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j(\arccos u)_j(\sqrt{1-u^2})_{k-j})$
$\arctan u$	$1+u^2$	$\frac{1}{1+(u)_0^2}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\arctan u)_j(1+u^2)_{k-j})$
$\mathrm{arccot}\, u$	$-(1+u^2)$	$\frac{-1}{1+(u)_0^2}((u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j(\mathrm{arccot}\, u)_j(1+u^2)_{k-j})$
$\tanh u$	$\cosh^2 u$	$\frac{1}{\cosh^2(u)_0}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\tanh u)_j(\cosh^2 u)_{k-j})$
$\coth u$	$-\sinh^2 u$	$\frac{-1}{\sinh^2(u)_0}((u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j(\coth u)_j(\sinh^2 u)_{k-j})$
$\mathrm{arsinh}\, u$	$\sqrt{u^2+1}$	$\frac{1}{\sqrt{(u)_0^2+1}}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\mathrm{arsinh}\, u)_j(\sqrt{u^2+1})_{k-j})$
$\mathrm{arcosh}\, u$	$\sqrt{u^2-1}$	$\frac{1}{\sqrt{(u)_0^2-1}}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\mathrm{arcosh}\, u)_j(\sqrt{u^2-1})_{k-j})$
$\mathrm{artanh}\, u$	$1-u^2$	$\frac{1}{1-(u)_0^2}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\mathrm{artanh}\, u)_j(1-u^2)_{k-j})$
$\mathrm{arcoth}\, u$	$1-u^2$	$\frac{1}{1-(u)_0^2}((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\mathrm{arcoth}\, u)_j(1-u^2)_{k-j})$

Table 4.1: Table of Taylor coefficients

In PASCAL-XSC there are also some exponential and logarithmic functions for special bases (2 and 10), for which we give the recursion formulas here, which are easily deduced from the above ones:

$$
\begin{aligned}
(2^u)_k &= \frac{\ln 2}{k}\sum_{j=0}^{k-1}(k-j)(2^u)_j(u)_{k-j} \\
(10^u)_k &= \frac{\ln 10}{k}\sum_{j=0}^{k-1}(k-j)(10^u)_j(u)_{k-j} \\
({}_2\log u)_k &= \frac{1}{\ln 2}(\ln u)_k \\
({}_{10}\log u)_k &= \frac{1}{\ln 10}(\ln u)_k
\end{aligned}
$$

Finally, we mention that also simple formulas for differentiation and integration can be derived which are even essentially the same as formula (5.12). From term-wise differentiation and integration of a Taylor series we see immediately that:

$$
(\frac{du}{dt})_k = (k+1)(u)_{k+1}, \quad k \geq 0 \tag{5.19}
$$

and

$$
(\int_{t_0}^{t} u(s)ds)_0 = 0, \quad (\int_{t_0}^{t} u(s)ds)_k = \frac{(u)_{k-1}}{k}, \quad k \geq 1 \ . \tag{5.20}
$$

We do not include these two operations in our module for the computation of Taylor coefficients. The reason for this is that in this module we want to include only operations which for a given set of indices $k = 0, \ldots, p$, compute the exact values of the Taylor coefficients (or enclosures of these values on the computer) for *all* of these indices. However, with differentiation, we are no longer able to compute the p-th coefficient, since to do this we would need the $(p+1)$-st coefficient of the original function which is not supplied in the program when the max. order is p.

5.1.3 Algorithms

We implement the recursive computation of the Taylor coefficients as a so-called Taylor arithmetic. I.e. we introduce (p+1)-tupels $((u)_0, \ldots, (u)_p)$ containing the Taylor coefficients of u up to the p-th order and define arithmetic operations and standard function for these tupels in such a way that the result-tupel of each operation contains the Taylor coefficients of the resulting function (up to the p-th order). In order to take the roundoff errors into account we use intervals for the representation of the Taylor coefficients such that we get guaranteed enclosures thereof.

For these $(p+1)$-tupels we use the PASCAL-XSC data type

```
TYPE  itaylor  = DYNAMIC ARRAY [*] OF INTERVAL;
```

The index range must always be $0, \ldots, p$, where $p \geq 0$ is some arbitrary integer which must be the same for all operands in a computation. However, none of the algorithms checks this condition, i.e. the algorithms may fail if they are used with operands of differing length.

We begin with the four basic operators $+, -, \cdot$ and $/$.

Algorithm 5.1: $+\,(u, v)$ {operator}

{Computation of the Taylor coefficients for $u + v$}
input: *itaylor* u, v of length $p + 1$
output: Enclosure of the Taylor coefficients of $u + v$ (operator result)

> **for** $k := 0$ **to** p **do** $w_k := u_k + v_k$
> **return** w

Algorithm 5.2: $-\,(u, v)$ {operator}

{Computation of the Taylor coefficients for $u - v$}
input: *itaylor* u, v of length $p + 1$
output: Enclosure of the Taylor coefficients of $u - v$ (operator result)

> **for** $k := 0$ **to** p **do** $w_k := u_k - v_k$
> **return** w

Algorithm 5.3: $*\,(u, v)$ {operator}

{Computation of the Taylor coefficients for $u \cdot v$}
input: *itaylor* u, v of length $p + 1$
output: Enclosure of the Taylor coefficients of $u \cdot v$ (operator result)

> **for** $k := 0$ **to** p **do** $w_k := \Diamond \sum_{j=0}^{k} u_j \cdot v_{k-j}$
> **return** w

Algorithm 5.4: $/\,(u, v)$ {operator}

{Computation of the Taylor coefficients for u/v}
input: *itaylor* u, v of length $p + 1$
output: Enclosure of the Taylor coefficients of u/v (operator result)

> **for** $k := 0$ **to** p **do** $w_k := \Diamond(u_k - \sum_{j=1}^{k} v_j \cdot w_{k-j})/v_0$
> **return** w

Square, squareroot and exponential are formulated along (5.9, 5.8) and (5.13).

Algorithm 5.5: Sqr (u) {function}

{Computation of the Taylor coefficients for u^2 }
input: *itaylor* u of length $p+1$
output: Enclosure of the Taylor coefficients of u^2 (function result)

$w_0 := u_0^2$
for $k := 1$ **to** p **do** $m := (k+1)$ **div** 2 {integer division}
$w_k := 2 \mathbin{\Diamond\!\!\!\!\cdot\,} \Diamond \sum_{j=0}^{m-1} u_j \cdot u_{k-j}$
if not $\mathrm{odd}(k)$ **then** $w_k := w_k + u_m^2$
Sqr $:= w$

Algorithm 5.6: Sqrt (u) {function}

{Computation of the Taylor coefficients for $\sqrt{u}$ }
input: *itaylor* u of length $p+1$
output: Enclosure of the Taylor coefficients of $\sqrt{u}$ (function result)

$w_0 := \sqrt{u_0}$
for $k := 1$ **to** p **do** $m := (k+1)$ **div** 2 {integer division}
$w_k := 2 \mathbin{\Diamond\!\!\!\!\cdot\,} \Diamond \sum_{j=1}^{m-1} w_j \cdot w_{k-j}$
if not $\mathrm{odd}(k)$ **then** $w_k := w_k + w_m^2$
$w_k := (u_k - w_k)/(2 \cdot w_0)$
Sqrt $:= w$

Algorithm 5.7: Exp (u) {function}

{Computation of the Taylor coefficients for e^u }
input: *itaylor* u of length $p+1$
output: Enclosure of the Taylor coefficients of e^u (function result)

$w_0 := \exp^{u_0}$
for $k := 1$ **to** p **do** $w_k := (\sum_{j=0}^{k-1} (k-j) \cdot w_j \cdot u_{k-j})/k$
Exp $:= w$

The next algorithm implements the pair of formulas (5.14) for the sine- and cosine-functions. It returns the Taylor coefficients for both functions and can therefore be used for each, sine and cosine.

Algorithm 5.8: Sin_Cos (u, w_sin, w_cos) {procedure}

{Computation of the Taylor coefficients for $\sin u$ and $\cos u$ }
input: *itaylor* u of length $p+1$
output: w_sin, enclosures of the Taylor coefficients of $\sin u$ and
w_cos, enclosures of the Taylor coefficients of $\cos u$

$w\_sin_0 := \sin(u_0)$
$w\_cos_0 := \cos(u_0)$

for $k := 1$ **to** p **do** $w\_sin_k := (\sum_{j=0}^{k-1}(k-j) \cdot w\_cos_j \cdot u_{k-j})/k$

$w\_cos_k := -(\sum_{j=0}^{k-1}(k-j) \cdot w\_sin_j \cdot u_{k-j})/k$

return w_sin, w_cos

Thus we have the following two simple algorithms for the sine and the cosine functions:

Algorithm 5.9: Sin (u) {function}

{Computation of the Taylor coefficients for $\sin u$}
input: *itaylor* u of length $p+1$
output: Enclosures of the Taylor coefficients of $\sin u$ (function result)

Sin_Cos (u, w_sin, w_cos)
Sin $:= w\_sin$

Algorithm 5.10: Cos (u) {function}

{Computation of the Taylor coefficients for $\cos u$}
input: *itaylor* u of length $p+1$
output: Enclosures of the Taylor coefficients of $\cos u$ (function result)

Sin_Cos (u, w_sin, w_cos)
Cos $:= w\_cos$

Precisely the same algorithms 5.5.8 – 5.5.10 can also be used for the hyperbolic sine and cosine functions. Only the minus sign in the second line of the for-loop in algorithm 5.5.8 has to be discarded (compare (5.14) and (5.15)).

The last two algorithms are the one for the power function u^a with constant exponent a,

Algorithm 5.11: Power (u) {function}

{Computation of the Taylor coefficients for u^a }
input: *itaylor* u of length $p+1$, *interval* a
output: Enclosure of the Taylor coefficients of u^a (function result)

$w_0 := \text{power}(u_0, a)$

for $k := 1$ to **to** p **do** $w_k := (\sum_{j=0}^{k-1}(a \cdot (k-j) - j) \cdot w_j \cdot u_{k-j})/(k \cdot u_0)$

Power $:= w$

and a generic algorithm for the remaining elementary functions, which implements (5.18) where for each specific function w the corresponding function g from Table 4.1 has to be inserted.

Algorithm 5.12: Function_f (u) {function}

{Computation of the Taylor coefficients for a function $f(u)$ with $f' = u'/g$ }
input: *itaylor* u of length $p+1$
output: Enclosure of the Taylor coefficients of $f(u)$ (function result)

{Compute Taylor coefficients of g, using the above algorithms}
$w_0 := f(u_0)$
for $k := 1$ **to** $p+1$ **do** $w_k := (u_k - (\sum_{j=1}^{k-1} j \cdot w_j \cdot g_{k-j})/k)/g_0$
Function_f $:= w$

5.1.4 PASCAL-XSC Program Code

Here we list the PASCAL-XSC module `itaylor` which implements the automatic computation of Taylor coefficients in interval arithmetic, i.e. with the operators and functions in this module we can compute guaranteed enclosures of Taylor coefficients. At the end of this section we indicate which changes have to be done in the module to adapt it to other data types.

```
MODULE itaylor;

USE GLOBAL i_ari;

{---------------------------------------------------------------------------}
{  This module itayor contains operators and functions for the recursive     }
{  computation of Taylor coefficients. These operators and functions are:    }
{    +, -(unary), +, -, *, /, sqr, sqrt, exp, exp2, exp10, ln, log2, log10,  }
{    power, sin, cos, tan, cot, arcsin, arccos, arctan, arccot,              }
{    sinh, cosh, tanh, coth, arsinh, arcosh, artanh, arcoth                  }
{---------------------------------------------------------------------------}
{  The data type for Taylor coefficients is a dynamic array of interval.     }
{  A variable of type itaylor represents the Taylor coeffcients, i.e.        }
{  u[0] = (u)_0, ..., u[n] = (u)_n; all operands are assumed to have the     }
{  same index range starting from 0. This is never checked however !         }
{---------------------------------------------------------------------------}

GLOBAL TYPE itaylor = GLOBAL DYNAMIC ARRAY [*] OF INTERVAL;

{---------------------------------------------------------------------------}
{ Define an itaylor-constant by assigning a value                            }
{---------------------------------------------------------------------------}

GLOBAL OPERATOR := ( VAR u : itaylor; c : INTERVAL );
VAR k : INTEGER;
BEGIN
   u[0]:= c;
   FOR k:= 1 TO ub(u) DO u[k]:= 0.0;
END;

GLOBAL OPERATOR := ( VAR u : itaylor; c : REAL );
VAR k : INTEGER;
BEGIN
   u[0]:= c;
   FOR k:= 1 TO ub(u) DO u[k]:= 0.0;
END;
```

```
{---------------------------------------------------------------------------}
{ Define expansion point x0 and make x the independent variable             }
{---------------------------------------------------------------------------}

GLOBAL PROCEDURE expand( VAR u : itaylor; u0 : INTERVAL );
VAR k : INTEGER;
BEGIN
   u[0]:= u0;
   IF ub(u) > 0 THEN  u[1]:= 1.0;
   FOR k:= 2 TO ub(u) DO u[k]:= 0.0;
END;

GLOBAL PROCEDURE expand( VAR u : itaylor; u0 : REAL );
VAR k : INTEGER;
BEGIN
   u[0]:= u0;
   IF ub(u) > 0 THEN  u[1]:= 1.0;
   FOR k:= 2 TO ub(u) DO u[k]:= 0.0;
END;

{---------------------------------------------------------------------------}
{ Monadic operators + and - for itaylor operands                            }
{---------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u : itaylor ) it_plus : itaylor[0..ub(u)];
BEGIN
   it_plus:=  u;
END;

GLOBAL OPERATOR - ( VAR u : itaylor ) it_minus : itaylor[0..ub(u)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO  it_minus[k]:= - u[k];
END;

{---------------------------------------------------------------------------}
{ Operators +, -, * and / for two itaylor operands                          }
{---------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u,v : itaylor ) itit_add : itaylor[0..ub(u)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO  itit_add[k]:= u[k] + v[k];
END;

GLOBAL OPERATOR - ( VAR u,v : itaylor ) itit_sub : itaylor[0..ub(u)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO  itit_sub[k]:= u[k] - v[k];
END;

GLOBAL OPERATOR * ( VAR u,v : itaylor ) itit_mul : itaylor[0..ub(u)];
VAR k,j : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO
       itit_mul[k]:= ##( FOR j:= 0 TO k SUM( u[j]*v[k-j] ) );
END;

GLOBAL OPERATOR / ( VAR u,v : itaylor ) itit_div : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    w   : itaylor[0..ub(u)];
BEGIN
   FOR k:= 0 TO ub(u) DO
       w[k]:= ##(  u[k] - ( FOR j:= 1 TO k SUM(v[j]*w[k-j]) ) ) / v[0];
```

```
   itit_div:= w;
END;

{---------------------------------------------------------------------------}
{ Operators +, -, * and / for one itaylor and one INTERVAL operand          }
{---------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u : itaylor; VAR v : INTERVAL )
                                          iti_add : itaylor[0..ub(u)];
BEGIN
   iti_add   := u;
   iti_add[0]:= u[0] + v;
END;

GLOBAL OPERATOR - ( VAR u : itaylor; VAR v : INTERVAL )
                                          iti_sub : itaylor[0..ub(u)];
BEGIN
   iti_sub   := u;
   iti_sub[0]:= u[0] - v;
END;

GLOBAL OPERATOR * ( VAR u : itaylor; VAR v : INTERVAL )
                                          iti_mul : itaylor[0..ub(u)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO  iti_mul[k]:= u[k]*v;
END;

GLOBAL OPERATOR / ( VAR u : itaylor; VAR v : INTERVAL )
                                          iti_div : itaylor[0..ub(u)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO  iti_div[k]:= u[k] / v;
END;

{---------------------------------------------------------------------------}
{ Operators +, -, * and / for one INTERVAL and one itaylor operand          }
{---------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u : INTERVAL; VAR v : itaylor )
                                          iit_add : itaylor[0..ub(v)];
BEGIN
   iit_add   := v;
   iit_add[0]:= u + v[0];
END;

GLOBAL OPERATOR - ( VAR u : INTERVAL; VAR v : itaylor )
                                          iit_sub : itaylor[0..ub(v)];
BEGIN
   iit_sub   := - v;
   iit_sub[0]:= u - v[0];
END;

GLOBAL OPERATOR * ( VAR u : INTERVAL; VAR v : itaylor )
                                          iit_mul : itaylor[0..ub(v)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(v) DO  iit_mul[k]:= u*v[k];
END;

GLOBAL OPERATOR / ( VAR u : INTERVAL; VAR v : itaylor )
                                          iit_div : itaylor[0..ub(v)];
VAR k,j : INTEGER;
    w   : itaylor[0..ub(v)];
```

```
BEGIN
   w[0]:= u / v[0];
   FOR k:= 1 TO ub(v) DO
       w[k]:= - ##( FOR j:= 1 TO k SUM( v[j]*w[k-j] ) ) / v[0];
   iit_div:= w;
END;

{----------------------------------------------------------------------------}
{ Operators +, -, * and / for one itaylor and one REAl operand               }
{----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u : itaylor; v : REAL )
                                         itr_add : itaylor[0..ub(u)];
BEGIN
   itr_add    := u;
   itr_add[0]:= u[0] + v;
END;

GLOBAL OPERATOR - ( VAR u : itaylor; v : REAL )
                                        itr_sub : itaylor[0..ub(u)];
BEGIN
   itr_sub    := u;
   itr_sub[0]:= u[0] - v;
END;

GLOBAL OPERATOR * ( VAR u : itaylor; v : REAL )
                                        itr_mul : itaylor[0..ub(u)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO  itr_mul[k]:= u[k]*v;
END;

GLOBAL OPERATOR / ( VAR u : itaylor; v : REAL )
                                        itr_div : itaylor[0..ub(u)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(u) DO  itr_div[k]:= u[k] / v;
END;

{----------------------------------------------------------------------------}
{ Operators +, -, * and / for one REAL and one itaylor operand               }
{----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( u : REAL; VAR v : itaylor )
                                        rit_add : itaylor[0..ub(v)];
BEGIN
   rit_add    := v;
   rit_add[0]:= u + v[0];
END;

GLOBAL OPERATOR - ( u : REAL; VAR v : itaylor )
                                        rit_sub : itaylor[0..ub(v)];
BEGIN
   rit_sub    := - v;
   rit_sub[0]:= u - v[0];
END;

GLOBAL OPERATOR * ( u : REAL; VAR v : itaylor )
                                        rit_mul : itaylor[0..ub(v)];
VAR k : INTEGER;
BEGIN
   FOR k:= 0 TO ub(v) DO  rit_mul[k]:= u*v[k];
END;
```

```
GLOBAL OPERATOR / ( u : REAL; VAR v : itaylor )
                                        rit_div : itaylor[0..ub(v)];
VAR k,j : INTEGER;
    w   : itaylor[0..ub(v)];
BEGIN
   w[0]:= u / v[0];
   FOR k:= 1 TO ub(v) DO
       w[k]:= - ##( FOR j:= 1 TO k SUM( v[j]*w[k-j] ) ) / v[0];
   rit_div:= w;
END;

{----------------------------------------------------------------------------}
{                 elementary functions for itaylor                           }
{----------------------------------------------------------------------------}

GLOBAL FUNCTION sqr( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j,m : INTEGER;
    w     : itaylor[0..ub(u)];
BEGIN
   w[0]:= sqr(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      m   := (k+1) DIV 2;
      w[k]:= 2.0 * ##( FOR j:= 0 TO m-1 SUM ( u[j] * u[k-j] ) );
      IF  NOT ODD(k)  THEN  w[k]:= w[k] + SQR( u[m] );
   END;
   sqr:= w;
END;

GLOBAL FUNCTION sqrt( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j,m : INTEGER;
    w     : itaylor[0..ub(u)];
BEGIN
   w[0]:= SQRT( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      m   := (k+1) DIV 2;
      w[k]:= 2.0 * ##( FOR j:= 1 TO m-1 SUM ( w[j]*w[k-j] )  );
      IF  NOT ODD(k)  THEN  w[k]:= w[k] + SQR( w[m] );
      w[k]:= ( u[k] - w[k] ) / ( 2.0*w[0] );
   END;
   sqrt:= w;
END;

GLOBAL FUNCTION power ( VAR u : itaylor; VAR a : INTER-
VAL ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    w   : itaylor[0..ub(u)];
BEGIN
   w[0]:= power( u[0], a );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 0 TO k-1 DO  w[k]:= w[k] + (a*(k-j)-j) * w[j] * u[k-j];
      w[k]:= w[k] / (k*u[0]);
   END;
   power:= w;
END;

GLOBAL FUNCTION power ( VAR u : itaylor; a : REAL ) : itaylor[0..ub(u)];
BEGIN
   power:= power( u, INTVAL(a) );
END;
```

```
GLOBAL FUNCTION exp( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    w   : itaylor[0..ub(u)];
BEGIN
   w[0]:= EXP( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 0 TO k-1 DO w[k]:= w[k]  +  (k-j) * w[j] * u[k-j];
      w[k]:= w[k] / k;
   END;
   exp:= w;
END;

GLOBAL FUNCTION exp2( VAR u : itaylor ) : itaylor[0..ub(u)];
{ exp2(x) = exp(x*ln(2)) }
VAR k,j : INTEGER;
    ln2 : INTERVAL;
    w   : itaylor[0..ub(u)];
BEGIN
   ln2 := LN(INTVAL(2.0));
   w[0]:= EXP2( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 0 TO k-1 DO w[k]:= w[k]  +  (k-j) * w[j] * u[k-j];
      w[k]:= ln2 * w[k] / k;
   END;
   exp2:= w;
END;

GLOBAL FUNCTION exp10( VAR u : itaylor ) : itaylor[0..ub(u)];
{ exp10(x) = exp(x*ln(10)) }
VAR k,j : INTEGER;
    ln10: INTERVAL;
    w   : itaylor[0..ub(u)];
BEGIN
   ln10:= LN(INTVAL(10.0));
   w[0]:= EXP10( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 0 TO k-1 DO w[k]:= w[k]  +  (k-j) * w[j] * u[k-j];
      w[k]:= ln10 * w[k] / k;
   END;
   exp10:= w;
END;

GLOBAL FUNCTION ln( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    w   : itaylor[0..ub(u)];
BEGIN
   w[0]:= LN( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO w[k]:= w[k]  +  j * w[j] * u[k-j];
      w[k]:= ( u[k] - w[k] / k ) / u[0];
   END;
   ln:= w;
END;

GLOBAL FUNCTION log2( VAR u : itaylor ) : itaylor[0..ub(u)];
{ log2(x) = ln(x)/ln(2) }
```

```
BEGIN
   log2:= ln(u) / LN(INTVAL(2.0));
END;

GLOBAL FUNCTION log10( VAR u : itaylor ) : itaylor[0..ub(u)];
{ log10(x) = ln(x)/ln(10) }
BEGIN
   log10:= ln(u) / LN(INTVAL(10.0));
END;

PROCEDURE sin_cos( VAR u,w_sin,w_cos : itaylor );
VAR k,j : INTEGER;
BEGIN
   w_sin[0]:= SIN( u[0] );
   w_cos[0]:= COS( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w_sin[k]:= 0.0;
      w_cos[k]:= 0.0;
      FOR j:= 0 TO k-1 DO w_sin[k]:= w_sin[k] + (k-j)*w_cos[j]*u[k-j];
      FOR j:= 0 TO k-1 DO w_cos[k]:= w_cos[k] + (k-j)*w_sin[j]*u[k-j];
      w_sin[k]:=   w_sin[k] / k;
      w_cos[k]:= - w_cos[k] / k;
   END;
END;

GLOBAL FUNCTION sin( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR w_sin,w_cos : itaylor[0..ub(u)];
BEGIN
   sin_cos( u, w_sin, w_cos );
   sin:= w_sin;
END;

GLOBAL FUNCTION cos( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR w_sin,w_cos : itaylor[0..ub(u)];
BEGIN
   sin_cos( u, w_sin, w_cos );
   cos:= w_cos;
END;

GLOBAL FUNCTION tan( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g   := sqr(cos(u));
   w[0]:= TAN(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   tan:= w;
END;

GLOBAL FUNCTION cot( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g   := sqr(sin(u));
   w[0]:= COT(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
```

```
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= - ( u[k] + w[k] / k ) / g[0];
   END;
   cot:= w;
END;

GLOBAL FUNCTION arcsin( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := sqrt(1.0-sqr(u));
   w[0]:= ARCSIN(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   arcsin:= w;
END;

GLOBAL FUNCTION arccos( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := -sqrt(1.0-sqr(u));
   w[0]:= ARCCOS(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   arccos:= w;
END;

GLOBAL FUNCTION arctan( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := 1.0 + sqr(u);
   w[0]:= ARCTAN( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   arctan:= w;
END;

GLOBAL FUNCTION arccot( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := -( 1.0 + sqr(u) );
   w[0]:= ARCCOT(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   arccot:= w;
```

```
END;

PROCEDURE sinh_cosh( VAR u,w_sinh,w_cosh : itaylor );
VAR k,j : INTEGER;
BEGIN
   w_sinh[0]:= SINH( u[0] );
   w_cosh[0]:= COSH( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w_sinh[k]:= 0.0;
      w_cosh[k]:= 0.0;
      FOR j:= 0 TO k-1 DO w_sinh[k]:= w_sinh[k] + (k-j)*w_cosh[j]*u[k-j];
      FOR j:= 0 TO k-1 DO w_cosh[k]:= w_cosh[k] + (k-j)*w_sinh[j]*u[k-j];
      w_sinh[k]:=  w_sinh[k] / k;
      w_cosh[k]:=  w_cosh[k] / k;
   END;
END;

GLOBAL FUNCTION sinh( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR w_sinh,w_cosh : itaylor[0..ub(u)];
BEGIN
   sinh_cosh( u, w_sinh, w_cosh );
   sinh:= w_sinh;
END;

GLOBAL FUNCTION cosh( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR w_sinh,w_cosh : itaylor[0..ub(u)];
BEGIN
   sinh_cosh( u, w_sinh, w_cosh );
   cosh:= w_cosh;
END;

GLOBAL FUNCTION tanh( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g   := sqr(cosh(u));
   w[0]:= TANH(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   tanh:= w;
END;

GLOBAL FUNCTION coth( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g   := sqr(sinh(u));
   w[0]:= COTH(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= - ( u[k] + w[k] / k ) / g[0];
   END;
   coth:= w;
END;

GLOBAL FUNCTION arsinh( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
```

```
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := sqrt(sqr(u)+1.0);
   w[0]:= ARSINH(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   arsinh:= w;
END;

GLOBAL FUNCTION arcosh( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := sqrt(sqr(u)-1.0);
   w[0]:= ARCOSH(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   arcosh:= w;
END;

GLOBAL FUNCTION artanh( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := 1.0 - sqr(u);
   w[0]:= ARTANH( u[0] );
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   artanh:= w;
END;

GLOBAL FUNCTION arcoth( VAR u : itaylor ) : itaylor[0..ub(u)];
VAR k,j : INTEGER;
    g,w : itaylor[0..ub(u)];
BEGIN
   g    := 1.0 - sqr(u);
   w[0]:= ARCOTH(u[0]);
   FOR k:= 1 TO ub(u) DO
   BEGIN
      w[k]:= 0.0;
      FOR j:= 1 TO k-1 DO  w[k]:= w[k]  +  j * w[j] * g[k-j];
      w[k]:= ( u[k] - w[k] / k ) / g[0];
   END;
   arcoth:= w;
END;

END.
```

Concluding this section we mention that by only a few simple changes the module can be converted to a module rtaylor for the computation of Taylor coefficients using the REAL data type, or to a module ctaylor using the COMPLEX data type or even to a module citaylor using the CINTERVAL data type (i.e. complex intervals).

Apart from renaming the module the following changes should be made:

- change the `USE`-clause; no `USE` is necessary for REAL, `USE c_ari` is needed for COMPLEX and `USE i_ari,c_ari,ci_ari` are needed for CINTERVAL.
- change all occurences of the data type INTERVAL to REAL, COMPLEX or CINTERVAL as desired. Then also the name of the data type itaylor should be changed into rtaylor, ctaylor or citaylor.
- change the `##` prefixes of the scalar product expressions to `#*` and discard the calls to the intrinsic function INTVAL in the cases of REAL and COMPLEX. In these cases also replace INTERVAL by REAL for the variables `ln2` and `ln10` in the functions `exp2` and `exp10`.
- operators with mixed operand types (taylor, REAL, INTERVAL, COMPLEX and/or CINTERVAL) may be added or deleted as desired.

There are however versions of PASCAL-XSC where many complex standard functions are not implemented (and also not those for complex intervals). With these versions only the basic arithmetic will work in complex, the standard functions will not.

5.1.5 Test Results

Example 5.1:

As a first test for the module itaylor we write a simple program which defines a function f using the type itaylor and evaluates it thereby using the operators and functions of the module. Thus the result is again an itaylor containing enclosures of the Taylor coefficients of f.

```
PROGRAM itseries;

USE itaylor;

{------------------------------------------------------------------------}
{ This program computes the Taylorpolynomial of a function f(x) up to     }
{ the term of p-th order using the operators and functions from           }
{ the module itaylor.                                                     }
{ The function f is defined as PASCAL-XSC function using the type itaylor. }
{------------------------------------------------------------------------}

FUNCTION f ( t : itaylor ) : itaylor[0..ub(t)];
BEGIN
   f:= exp( -sqr(t) )
END;

PROCEDURE main( p : INTEGER );
VAR k   : INTEGER;
```

```
     x,y : itaylor[0..p];
     x0  : INTERVAL;
  BEGIN
     WRITE( 'point of expansion : ' );  READ ( x0 );
     expand( x, x0 );
     y:= f( x );
     FOR k:= 0 TO p DO  WRITELN( k:3,' :  ', y[k] );
  END;

  VAR p : INTEGER;

  BEGIN
     REPEAT
        WRITE( 'order of Taylorpolynomial : ' ); READ ( p );
        IF p >= 0 THEN main ( p );
     UNTIL p < 0;
  END.
```

If we enter 10 for the order of the Taylor polynomials and 0, 1 and 10 for the expansion points then the program produces the following output for the function $f(t) = e^{-t^2}$ (we have added the user's input for completeness; entering -1 for the order at the end stops the program):

```
order of Taylorpolynomial : 10
point of expansion : 0
  0 :   [  1.000000000000000E+000,  1.000000000000000E+000 ]
  1 :   [  0.000000000000000E+000,  0.000000000000000E+000 ]
  2 :   [ -1.000000000000000E+000, -1.000000000000000E+000 ]
  3 :   [  0.000000000000000E+000,  0.000000000000000E+000 ]
  4 :   [  5.000000000000000E-001,  5.000000000000000E-001 ]
  5 :   [  0.000000000000000E+000,  0.000000000000000E+000 ]
  6 :   [ -1.666666666666667E-001, -1.666666666666666E-001 ]
  7 :   [  0.000000000000000E+000,  0.000000000000000E+000 ]
  8 :   [  4.166666666666666E-002,  4.166666666666668E-002 ]
  9 :   [  0.000000000000000E+000,  0.000000000000000E+000 ]
 10 :   [ -8.333333333333335E-003, -8.333333333333331E-003 ]
order of Taylorpolynomial : 10
point of expansion : 1
  0 :   [  3.678794411714422E-001,  3.678794411714424E-001 ]
  1 :   [ -7.357588823428847E-001, -7.357588823428845E-001 ]
  2 :   [  3.678794411714422E-001,  3.678794411714424E-001 ]
  3 :   [   2.45252960780961E-001,   2.45252960780962E-001 ]
  4 :   [ -3.065662009762021E-001, -3.065662009762017E-001 ]
  5 :   [   2.45252960780960E-002,   2.45252960780963E-002 ]
  6 :   [   9.40136349660351E-002,   9.40136349660354E-002 ]
  7 :   [  -3.38682660126091E-002,  -3.38682660126089E-002 ]
  8 :   [  -1.50363422383566E-002,  -1.50363422383565E-002 ]
  9 :   [   1.08676907224367E-002,   1.08676907224369E-002 ]
 10 :   [    8.3373030318394E-004,    8.3373030318397E-004 ]
order of Taylorpolynomial : 10
point of expansion : 10
  0 :   [  3.720075976020835E-044,  3.720075976020837E-044 ]
  1 :   [ -7.440151952041674E-043, -7.440151952041670E-043 ]
  2 :   [  7.402951192281460E-042,  7.402951192281467E-042 ]
  3 :   [ -4.885699781840701E-041, -4.885699781840694E-041 ]
  4 :   [   2.40583513495893E-040,   2.40583513495895E-040 ]
  5 :   [  -9.42791254856215E-040,  -9.42791254856212E-040 ]
  6 :   [   3.06244301168874E-039,   3.06244301168876E-039 ]
  7 :   [  -8.48046824629466E-039,  -8.48046824629463E-039 ]
  8 :   [   2.04355598628143E-038,   2.04355598628145E-038 ]
  9 :   [  -4.35278067515223E-038,  -4.35278067515220E-038 ]
 10 :   [   8.29685015304811E-038,   8.29685015304817E-038 ]
order of Taylorpolynomial : -1
```

It should be noted again that these numbers are *guaranteed enclosures* of the *exact* Taylor coefficients of the function f in the program.

Example 5.2:

In the second example we use the operations from the itaylor module plus an additional function `integral` which computes the Taylor coefficients of the integral $\int_{t_0}^{t} u(s)ds$ of a function $u(t)$ according to (5.20).

The program computes the Taylor polynomial of the solution of an initial value problem with a first order ordinary differential equation about the expansion point $t = t_0$ by use of Picard iteration. The initial value problem is

$$u'(t) = f(t, u(t)), \quad u(t_0) = u_0 \ . \tag{5.21}$$

Starting with a constant function $u_0(t) = u_0$ we compute p iterations with the equivalent integral equation:

$$u_i(t) = u_0 + \int_{t_0}^{t} f(s, u_{i-1}(s))ds \ . \tag{5.22}$$

In this iteration the function f is evaluated by use of the operators and functions from the module itaylor and the integral by (5.20). Thus the result of each iteration in the program is the Taylor polynomial of degree p of the corresponding exact iterate. Since each iteration produces successively one additional term of the Taylor series of the solution $u(t)$ we have after p iterations the exact Taylor polynomial of $u(t)$, i.e. from the program we get enclosures for the first p coefficients of the Taylor series of the solution $u(t)$ of (5.21).

```
PROGRAM ode_series;

USE itaylor;

{------------------------------------------------------------------------}
{ This program computes an enclosure for the Taylorpolynomial of order p   }
{ of a function u(t) defined by a first order ode.                         }
{ For this purpose the operators and functions from the module itaylor     }
{ are used plus an additional function 'integral' which computes the       }
{ Taylor coefficients of the integral of a function.                       }
{ The right hand side f(t,u) of the ode is defined as PASCAL-XSC function  }
{ using the type itaylor.                                                  }
{------------------------------------------------------------------------}

FUNCTION integral ( u : itaylor ) : itaylor[0..ub(u)];
{ --- compute Taylorcoefficients of the integral of u }
VAR k : INTEGER;
BEGIN
   integral[0]:= 0.0;
   FOR k:= 1 TO ub(u) DO integral[k]:= u[k-1] / k;
END;
```

```
FUNCTION f ( t,u : itaylor ) : itaylor[0..ub(u)];
{ --- right hand side of differential equation }
BEGIN
   f:= exp(t) + t*cos(u);
END;

PROCEDURE main( p : INTEGER );
VAR i,k    : INTEGER;
    t,u    : itaylor[0..p];
    t0,u0  : INTERVAL;
BEGIN
   WRITE( 'enter initial point t0 : ' );  READ ( t0 );
   WRITE( 'enter initial value u0 : ' );  READ ( u0 );
   { --- define point of expansion and initial value }
   expand( t, t0 );
   u:= u0;
   { --- compute first p Taylor coefficients iteratively }
   FOR i:= 1 TO p DO u:= u0 + integral( f(t,u) );
   { --- print out Taylor coefficients }
   FOR k:= 0 TO p DO WRITELN( k:3,' :  ', u[k] );
END;

VAR p : INTEGER;

BEGIN
   REPEAT
      WRITE( 'order of Taylorpolynomial : ' ); READ ( p );
      IF p >= 0 THEN main ( p );
   UNTIL p < 0;
END.
```

In the program we use as an example the differential equation

$$u'(t) = e^t + t \cos u(t) \ .$$

The initial point t_0 and the initial value u_0 are supplied by the user. A sample output of the program is as follows, where the input -1 for the order of the Taylor polynomial stops the loop in the main program.

```
order of Taylorpolynomial : 20
enter initial point t0 : 0
enter initial value u0 : 0
  0 :  [  0.000000000000000E+000,  0.000000000000000E+000 ]
  1 :  [  1.000000000000000E+000,  1.000000000000000E+000 ]
  2 :  [  9.999999999999998E-001,  1.000000000000000E+000 ]
  3 :  [  1.666666666666666E-001,  1.666666666666667E-001 ]
  4 :  [ -8.333333333333335E-002, -8.333333333333331E-002 ]
  5 :  [ -1.916666666666667E-001, -1.916666666666665E-001 ]
  6 :  [  -1.027777777777778E-001,  -1.027777777777777E-001 ]
  7 :  [   1.21031746031745E-002,   1.21031746031747E-002 ]
  8 :  [  6.721230158730152E-002,  6.721230158730163E-002 ]
  9 :  [   5.95706569664902E-002,   5.95706569664904E-002 ]
 10 :  [   1.55285493827160E-002,   1.55285493827161E-002 ]
 11 :  [  -2.10813241542409E-002,  -2.10813241542407E-002 ]
 12 :  [  -3.03950091189675E-002,  -3.03950091189673E-002 ]
 13 :  [  -1.64632934902033E-002,  -1.64632934902031E-002 ]
 14 :  [    3.3958146451945E-003,    3.3958146451947E-003 ]
 15 :  [   1.41046457194150E-002,   1.41046457194151E-002 ]
 16 :  [   1.18691142536602E-002,   1.18691142536604E-002 ]
 17 :  [    2.5225477697761E-003,    2.5225477697763E-003 ]
 18 :  [  -5.41088587507608E-003,  -5.41088587507603E-003 ]
 19 :  [  -7.16313548253303E-003,  -7.16313548253297E-003 ]
 20 :  [   -3.5829610848609E-003,   -3.5829610848608E-003 ]
order of Taylorpolynomial : -1
```

As in the previous example the enclosures are *guaranteed enclosures* of the *exact* Taylor coefficients of the solution $u(t)$, however, we should bear in mind that even the exact Taylor polynomial is still only an approximation of the true solution $u(t)$. To get an enclosure of the true solution on some interval $[t_0, t_0 + h]$ we must also compute an enclosure of the corresponding remainder term. This can again be done with the operations in the module itaylor, but to this end we need somewhat more theory since a stepsize h and a first rough enclosure $[u^{(0)}]$ of u on $[t_0, t_0 + h]$ must be computed. For details see e.g. [247], [248], [207], [228], [229], [232].

We should also note that the computation of the Taylor coefficients in this example can be organized more efficiently: In the k-th iteration the coefficients $(u)_0, \ldots, (u)_{k-1}$ of the exact solution $u(t)$ are known already and $(u)_{k+1}, \ldots (u)_p$ can not yet be computed. Therefore it is cheaper to compute *only* the k-th coefficient in the k-th iteration and not all p coefficients which would mean that we unnecessarily recompute the coefficients with low indices and compute values for coefficients with high indices which are not yet the wanted exact Taylorcoefficients of u.

Further examples include problems where a guaranteed enclosure of a solution can be computed by use of a remainder term containing higher derivatives of a function at some unknown intermediate point of an interval. The enclosure of the solution of an initial value problem is such an example, another very important one is numerical quadrature. In Chapter 10 we will make intensive use of the module itaylor to enclose the remainder terms occuring there in order to compute enclosures for the values of definite integrals.

5.1.6 Notes and References

First we make some comments on the itaylor module and on possible variations and improvements thereof. The possible change of the data type interval to real, complex or complex interval has already been discussed in Section 5.1.4.

In order to make the use of the module more resistant against program crashes, it may be desirable to include checks for the index ranges in each operator. This will slow down the execution time only little, but it will be of great help especially in the phase of program development.

In the straightforward way the module is written it may be inefficient to use in certain special applications. As we saw in the last example, there the module could be used more efficiently if we had the possibility to modify the index range of the Taylor coefficients to be computed. This can be achieved in several ways like e.g. introducing global variables in the module which indicate the index range of the coefficients to be computed. Another possibility would be to integrate this information into the data type itaylor, e.g. by using a suitable record or taking additional array components to hold this information. Similarly a result length could be introduced indicating for the resulting Taylor polynomial of each operation up to which index the series could actually be computed (e.g. if an error occurred in the computation or operands with a differing number of terms were supplied, or if an operation like differentiation, (5.19), should be used also). An additional advantage of such a modification would be that monomials or polynomials with only few coefficients differing from zero could be handled considerably faster. For such an approach see e.g. [88].

In the case of division u/v the operator for itaylor/itaylor breaks down if both Taylor polynomials u in the numerator and v in the denominator start with a constant term equal to zero. However, often we still can compute a correct result by applying de L'Hospitals rule, i.e. by differentiating u and v and taking the quotient u'/v' as operands for the division. If necessary the differentiation has to be performed several times. However, it should be noted that in this case we can no longer compute the same number of Taylor coefficients for u/v as we had in the operands u and v. Hence an indication of the actual length of the resulting series should be generated as dicussed in the previous paragraph.

Automatic differentiation for the computation of Taylor coefficients was already used by R.E. Moore in 1965 for the solution of ordinary initial value problems. Later many important contributions came from Rall, Corliss, Griewank, Iri, Fischer and others. A recent collection of research and overview papers as well as an extended bibliography is contained in [115].

5.2 Functions in Several Variables, Gradients

5.2.1 Introduction

For a one dimensional function the computation of first order derivatives is a trivial special case of the computation of Taylor coefficients as we have done it in the previous Section. For a function in several variables there is in principle no difference also since we have to follow exactly the same pattern and repeat it for each of the independent variables, i.e. we have to implement a Taylor arithmetic which computes the first derivatives with respect to each independent variable and stores it in a vector. This so-called *forward mode* of automatic differentiation for gradients has been discussed in many places already, see [247], [268], [124] and others.

However, in this forward mode the cost for computing a value $f(x)$ and its gradient ∇f is about $n+1$ times higher than the cost for computing $f(x)$ alone, since each (scalar) operation f_k in f is now replaced by a vector operation involving f_k and ∇f_k.

Approximately the same increase in cost comes also with the computation of finite differences as an approximation for ∇f.

Therefore it is astonishing that a method exists which is able to deliver ∇f at a cost which is bounded by a *constant* multiple of the cost for computing f alone – i.e. this constant is independent of the number n of independent variables in the vector x. The value of this constant is less than 5, see e.g. [114], [104], [148], and in most cases the computation of f and its gradient ∇f turns out to be only little more than twice as expensive as the computation of f alone.

This enormous reduction of computational complexity is achieved by a very economic way of the application of the chain rule. However, the price we have to pay for this fast method, the so-called *reverse mode* of automatic differentiation, is that we have to store each intermediate result of the computation of f until the process of determining ∇f is completed. For complicated functions f with a large number of variables i.e. when x has very high dimension this may be a considerable amount of storage.

5.2.2 Theoretical Background

Let $x \in R^n$ be a vector of independent variables and $f(x)$ an expression whose value and whose gradient $\nabla f(x)$ are sought. We decompose f into its individual operations f_k in the following way.

First we identify the n independent variables $x_1, \dots, x_n$ in x as $f_1, \dots, f_n$:

$$\begin{array}{rcl} f_1 &=& x_1 \\ &\vdots& \\ f_n &=& x_n \ . \end{array} \tag{5.23}$$

Next, we assume that f contains m constants $c_1, \dots, c_m$, which we call

$$\begin{array}{rcccl} f_{n+1} &=& c_1 &=& x_{n+1} \\ &\vdots&&& \\ f_{n+m} &=& c_m &=& x_{n+m} \ . \end{array} \tag{5.24}$$

The remaining f_k's are now the operations which are actually performed in f:

$$\begin{array}{rcccl} f_{n+m+1} &=& f_{n+m+1}(x_1, \dots, x_{n+m}) &=& x_{n+m+1}(x_1, \dots, x_{n+m}) \\ &\vdots&&& \\ f = f_N &=& f_N(x_1, \dots, x_{N-1}) &=& x_N(x_1, \dots, x_{N-1}) \ . \end{array} \tag{5.25}$$

The values $x_{n+m+1}, \dots, x_N$ are the intermediate values which are obtained as the expression is computed and the final value, x_N is the value of f. All other f_k in (5.25) usually depend only on one or two of the previous x_k's, so the system in (5.23) – (5.25) is extremely sparse. The operations appearing as $f_k,\ k = n + m + 1, \dots, N$ are typically the arithmetic operations $+, -, *, /$ in which case f_k depends only on two variables or they are calls to standard functions like $\sin, \cos, \exp$, etc. in which case f_k even depends on one variable only. This decomposed form (5.23) – (5.25) is sometimes called the *code list* of f, see e.g. [114], [104]. Of course the decomposition into such a code list is not unique, in fact even for simple expressions there are usually many possibilities to generate a code list.

Now we associate with each x_k an *adjoint value* g_k which is the derivative of the final value $f = x_N = f_N$ with respect to the variable x_k:

$$g_k := \frac{\partial x_N}{\partial x_k}, \quad k = 1, \dots, N \ . \tag{5.26}$$

Obviously we have

$$g_N = 1 \quad \text{and} \quad \nabla f = (g_1, \ldots, g_n) \ . \tag{5.27}$$

From the chain rule we also get immediately

$$\begin{aligned} g_k &= \frac{\partial x_N}{\partial x_k} = (\frac{\partial f_N}{\partial x_1}, \ldots, \frac{\partial f_N}{\partial x_{N-1}}) \cdot (\frac{\partial x_1}{\partial x_k}, \ldots, \frac{\partial x_{N-1}}{\partial x_k}) \\ &= \frac{\partial f_N}{\partial x_k} + \sum_{i=k+1}^{N-1} \frac{\partial f_N}{\partial x_i} \cdot \frac{\partial x_i}{\partial x_k} \\ &= \sum_{i=k+1}^{N} \frac{\partial f_i}{\partial x_k} \cdot g_i \ . \end{aligned} \tag{5.28}$$

since $x_1, \ldots, x_{k-1}$ do not depend on x_k and $\partial x_k / \partial x_k = 1$.

This is a recursive formula for the adjoint values g_k which can be applied backwards starting from $k = N$ with $g_N = 1$ and then evaluating g_k for k down to 1. Therefore the method is also called the *reverse mode* of automatic differentiation.

The last sum in (5.28) has the following meaning: The k-th adjoint value g_k is obtained by accumulating all products $\partial f_i / \partial x_k \cdot g_i$ from those operations f_i in which x_k appears as an operand. If x_k does not appear in f_i then trivially $\partial f_i / \partial x_k = 0$.

We see that the computation of f and ∇f can be divided into two phases:

The forward sweep:
This is the computation of the function value f starting with x_1 at the beginning of the code list and ending with the final value $x_N = f_N = f$.

The reverse sweep:
This is the computation of the adjoint values g_k starting with $g_N = 1$ at the end of the code list and stepping backwards through the list ending with the final value g_1. At the end of this reverse sweep the components of ∇f can be found in $g_1, \ldots, g_N$.

The partial derivatives $\partial f_i / \partial x_k$ must be calculated for each operation f_i. Since for the f_i we use only a relatively small set of operations i.e. the arithmetic operations and the usual standard functions, these partial derivatives can be calulated and programmed once just as in the case of a forward method (see e.g. the Taylor arithmetic in the previous section 5.1). In the following table we summarize these derivatives for the basic arithmetic operations:

f_i	$\frac{\partial f_i}{\partial x_k}$	f_i	$\frac{\partial f_i}{\partial x_k}$
$x_k + x_r$	1	$x_k \cdot x_r$	x_r
$x_l + x_k$	1	$x_l \cdot x_k$	x_l
$x_k - x_r$	1	$\frac{x_k}{x_r}$	$\frac{1}{x_r}$
$x_l - x_k$	-1	$\frac{x_l}{x_k}$	$\frac{-f_i}{x_k}$

(5.29)

Table 4.2: Derivatives for the basic arithmetic operations

Similarly we get the next table for the derivatives $\partial f_i/\partial x_k$ if f_i is a standard function:

f_i	$\frac{\partial f_i}{\partial x_k}$	f_i	$\frac{\partial f_i}{\partial x_k}$	f_i	$\frac{\partial f_i}{\partial x_k}$
x_k^2	$2x_k$	$\sin x_k$	$\cos x_k$	$\sinh x_k$	$\cosh x_k$
$\sqrt{x_k}$	$\frac{1}{2f_i}$	$\cos x_k$	$-\sin x_k$	$\cosh x_k$	$\sinh x_k$
e^{x_k}	f_i	$\tan x_k$	$1+f_i^2$	$\tanh x_k$	$1-f_i^2$
2^{x_k}	$f_i \ln 2$	$\cot x_k$	$-(1+f_i^2)$	$\coth x_k$	$-(1-f_i^2)$
10^{x_k}	$f_i \ln 10$	$\arcsin x_k$	$\frac{1}{\sqrt{1-x_k^2}}$	$\operatorname{arsinh} x_k$	$\frac{1}{\sqrt{x_k^2+1}}$
$\ln x_k$	$\frac{1}{x_k}$	$\arccos x_k$	$\frac{-1}{\sqrt{1-x_k^2}}$	$\operatorname{arcosh} x_k$	$\frac{1}{\sqrt{x_k^2-1}}$
${}_2\log x_k$	$\frac{1}{x_k \ln 2}$	$\arctan x_k$	$\frac{1}{1+x_k^2}$	$\operatorname{artanh} x_k$	$\frac{1}{1-x_k^2}$
${}_{10}\log x_k$	$\frac{1}{x_k \ln 10}$	$\operatorname{arccot} x_k$	$\frac{-1}{1+x_k^2}$	$\operatorname{arcoth} x_k$	$\frac{1}{1-x_k^2}$

(5.30)

Table 4.3: Derivatives for standard functions

It is obvious how the computation of f using the code list and the forward sweep has to be done. Less obvious is how the reverse sweep should be organized even knowing all the partial derivatives from the tables above. In literature we can find many different ways to implement the reverse sweep, see e.g. [114], [104] and others. Here, we will discuss a very intuitive way for the realization of the reverse sweep which, in addition, is also very easy to implement in PASCAL-XSC.

Instead of using the code list directly we rather construct an equivalent representation of f in the form of a *computational graph* which is similar to a binary tree. Each node of this graph represents one of the operations f_i, i.e. some nodes represent just the independent variables $x_1, \ldots, x_n$ and the constants $c_1 = x_{n+1}, \ldots, c_m = x_{n+m}$. In the node representing f_i we store the corresponding values x_i and g_i as well as the information what operation has to be done in this node and pointers to the operands. We write the final node for $f = f_N = x_N$ at the top and put the others below recursively drawing a line from an upper node to a lower node if the upper one uses the lower one as an operand.

With this representation the forward sweep and the reverse sweep both can be carried out by recursively traversing the graph and thereby computing the desired values. In the forward sweep we compute the values x_i starting from the known values $x_1, \ldots, x_{n+m}$ at the bottom of the graph and going upwards ending with $f = f_N = x_N$ at the top of the graph. In addition, while carrying out the forward sweep we initialize the adjoint values g_i in each node with zero such that in the reverse sweep we can accumulate these quantities without any further initializations. The reverse sweep is carried out in the other direction, starting at the top of the graph with $g_N = 1$ and computing recursively the other g-values in the following way: in the node corresponding to f_i compute $\partial f_i / \partial x_k$ for each operand occurring in this node (there are at most two of them) and add the product of this derivative with the g-value g_i of the current node to the g-value g_k of the operand-node f_k

As an example for the computation of f and the adjoint values g_k in a computational graph consider the simple function

$$f(x_1, x_2, x_3) = x_1 x_2 x_3$$

from $I\!R^3 \to I\!R$.

We decompose f into the following code list:

$$\begin{array}{rcrcrcl} & & f_1 & = & x_1 & & \\ & & f_2 & = & x_2 & & \\ & & f_3 & = & x_3 & & \\ & & f_4 & = & x_4 & = & x_2 x_3 \\ f & = & f_5 & = & x_5 & = & x_1 x_4 \end{array}$$

The computational graph corresponding to this code list is shown in the following figure. Also the values of all intermediate results x_i and g_i are denoted there. f and ∇f are computed for the argument $x = (3, -2, 4)$. Each node f_i is drawn by use of a box containing three lines, the first line indicating the operation corresponding to this node, the second and third line containing the values of x_i and g_i for this node.

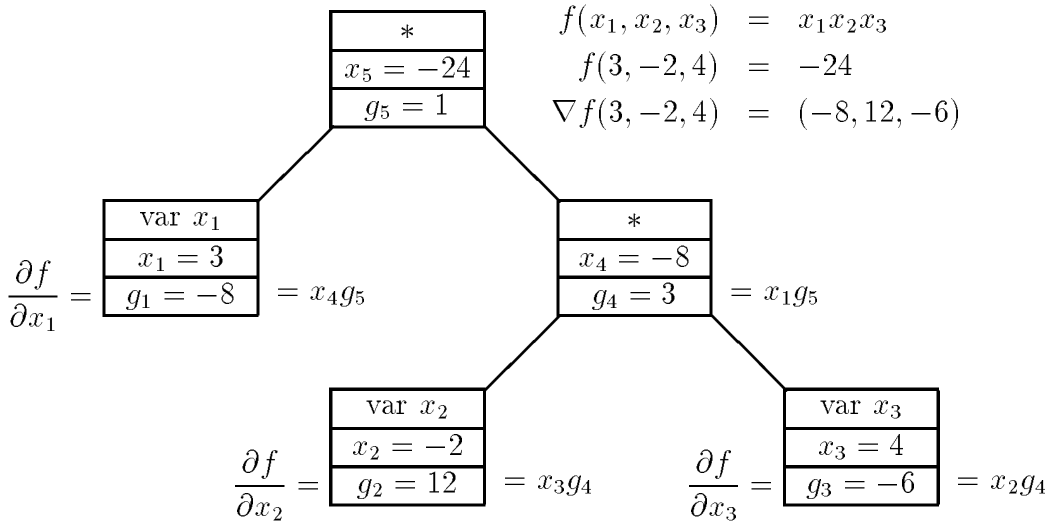


Figure 5.1: any comment ?

5.2.3 Algorithms

In this section we discuss the forward and the reverse sweep in a more algorithmic setting from which we will be able to write our PASCAL-XSC module.

We will formulate the forward sweep as a collection of operators and functions each of which implements a basic operation f_i. In each such operation a node of the computational tree will be generated and its components will be initialized properly:

- A component `op` contains the operation which has to be performed in this node.
- Two components `l` and `r` are pointers to the two operands (if there is only one operand, then `r` is not used).
- Two components `x` and `g` hold the function value x_i of the current node which is initialized with this value already at node generation and the adjoint value g_i which is initialized with zero at node generation time. The proper adjoint values will be accumulated in `g` later when the reverse sweep is carried out.
- A component `sub` which counts the references to this node from other nodes. I.e. `sub` indicates how often the current node appears as a subexpression in f.

Among the algorithms for the forward sweep there is a special one, `newf`, which is dedicated to the generation and initialization of a node for the computational graph. All the other ones call `newf` with suitable parameters to generate their specific node. These algorithms are very simple, in fact they are almost identical with each other differing only in the kind of operation which they perform. Therefore we only list three algorithms: the special algorithm `newf`, one example for a binary operator and another example for a standard function. The complete set of algorithms is listed in the program text in Section 5.2.4.

Algorithm 5.13: $\mathsf{Newf}\,(op, a, b, x)$ {function}

{Generation of a formula node}
input: Operation code op, formula pointers a, b, interval x (function value of the node)
output: Pointer to the node generated (as function value)

(a) new(f) {allocate new node f}

(b) {initialize node f}
$f \uparrow .op := op$
$f \uparrow .l := a$
$f \uparrow .r := b$
$f \uparrow .x := x$
$f \uparrow .g := 0.0$
$f \uparrow .sub := -1$ {subexpression counter}

(c) $\mathsf{Newf} := f$ {return pointer to generated node}

Algorithm 5.14: $+\,(a, b)$ {operator}

{Add two formula nodes}
input: Formula pointers a, b
output: Pointer to the node generated (operator result)

(a) {increase reference counter in operands}
$a \uparrow .sub := a \uparrow .sub + 1$
$b \uparrow .sub := b \uparrow .sub + 1$

(b) {do operation}
return $\mathsf{Newf}\,(op\_plus, a, b, a \uparrow .x + b \uparrow .x)$

Algorithm 5.15: $\mathsf{Sin}\,(a)$ {function}

{Sine of a formula node}
input: Formula pointer a
output: Pointer to the node generated (function result)

(a) {increase reference counter in operand}
$a \uparrow .sub := a \uparrow .sub + 1$

(b) {do operation}
$\mathsf{Sin} := \mathsf{Newf}\,(op\_sin, a, NIL, \sin(a \uparrow .x))$

In contrast the reverse sweep consists of just one function, `eval`, which by use of a local procedure traverses the graph recursively and accumulates the adjoint values in all nodes. Depending on the kind of operation in node f_i the product $\partial f_i / \partial x_k \cdot g_i$ is determined in a case-statement and added to g_k of the operand node f_k. In the following algorithm we do not list each of the different cases since as in the forward sweep they are all very similar and are listed in detail in the program text. The local procedure `reverse_sweep` is as follows.

Algorithm 5.16: ReverseSweep (f) {procedure}

{Reverse sweep for a formula f, local procedure to EVAL }
input: formula pointer f
output: none
side effects: computes all adjoint values in the graph of f, disposes the storage of the graph of f after this computation

(a) {accumulate $\partial f_i / \partial x_k \cdot g_i$ into g_k of the operands}
case $f \uparrow .op$ **of**

constant: nothing to do
variable: nothing to do
plus: $f \uparrow .l \uparrow .g := f \uparrow .l \uparrow .g + f \uparrow .g$
$f \uparrow .r \uparrow .g := f \uparrow .r \uparrow .g + f \uparrow .g$
minus: $f \uparrow .l \uparrow .g := f \uparrow .l \uparrow .g + f \uparrow .g$
$f \uparrow .r \uparrow .g := f \uparrow .r \uparrow .g - f \uparrow .g$
mult: $f \uparrow .l \uparrow .g := f \uparrow .l \uparrow .g + f \uparrow .g \cdot f \uparrow .r \uparrow .x$
$f \uparrow .r \uparrow .g := f \uparrow .r \uparrow .g + f \uparrow .g \cdot f \uparrow .l \uparrow .x$
div: $f \uparrow .l \uparrow .g := f \uparrow .l \uparrow .g + f \uparrow .g / f \uparrow .r \uparrow .x$
$f \uparrow .r \uparrow .g := f \uparrow .r \uparrow .g - f \uparrow .g \cdot f \uparrow .x / f \uparrow .r \uparrow .x$
sin: $f \uparrow .l \uparrow .g := f \uparrow .l \uparrow .g + f \uparrow .g \cdot \cos(f \uparrow .l \uparrow .x)$
$\vdots$ similarly for all other standard functions

end case

(b) {call ReverseSweep recursively for left operand}
if $f \uparrow .l \neq$ NIL **then**
if $f \uparrow .l \uparrow .sub \leq 0$ **then** ReverseSweep ($f \uparrow .l$)
else $f \uparrow .l \uparrow .sub := f \uparrow .l \uparrow .sub - 1$

(c) {call ReverseSweep recursively for right operand}
if $f \uparrow .r \neq$ NIL **then**
if $f \uparrow .r \uparrow .sub \leq 0$ **then** ReverseSweep ($f \uparrow .r$)
else $f \uparrow .r \uparrow .sub := f \uparrow .r \uparrow .sub - 1$

(d) {node f is no longer needed, if not independent variable then free used memory}
if $f \uparrow .op \neq$ variable **then** dispose(f)

The function `eval` contains a local recursive procedure `reverse_sweep` which actually carries out the reverse sweep. `eval` itself does nothing else but returning the value of f which has been computed in the forward sweep, initializing the adjoint value $g_N := 1$ and starting the reverse sweep by a call to the local procedure `reverse_sweep`.

Algorithm 5.17: Eval (f) {function}

{Evaluation and reverse sweep for a formula f }
input: formula pointer f

output: interval containing the value of f (function result)
side effects: computes all adjoint values in the graph of f, disposes the storage of the graph of f after this computation

(a) **return** $f\uparrow.x$ {has been computed in forward sweep}

(b) {initiate reverse sweep}
$f\uparrow.g := 1$
ReverseSweep (f) {call local recursive procedure}

5.2.4 PASCAL-XSC Program Code

The following PASCAL-XSC module `igradrev` contains an implementation of the reverse mode of automatic differentiation as described in the previous sections.

The data type `formula` is used to build the computational graph representing the formulas to be differentiated and is explained in detail in the comment at the beginning of the program. It is a record containing a component `op` which indicates the operation to be done in the current node. Further components are pointers to the left and right operands (`l`,`r`), the function value `x` and the adjoint value `g` of the current node. The component `sub` has a less obvious meaning: it is used to count the references from outside to the current node. In other words the value of `sub` shows how often the partial formula starting at the current node is used as an operand in different operations i.e. how often it appears as a subexpression in f.

This is an important information for the reverse sweep since according to (5.28) the adjoint value `g`$= g_k$ in each node f_k is valid only if *all* products $\partial f_i/\partial x_k \cdot g_i$ from *all* operations f_i using x_k are accumulated in `g`$= g_k$.

The basic philosophy underlying this implementation is as follows. For each expression which has to be evaluated the computational graph is constructed by use of the operators and the data type `formula`, i.e. the user writes the expression for f as an ordinary PASCAL-XSC expression using the data type `formula` which automatically generates the computational graph. Simultaneously with the construction of this graph the value of the expression is computed and each intermediate result is stored in the corresponding node of the graph. This phase of graph construction and computation of the function value is the forward sweep.

The backward sweep which computes all adjoint values in the graph is initiated by a call to function `eval`. When this function is finished with the computation of the adjoint values all nodes of the graph are disposed with the exception of the nodes which represent the independent variables. Therefore, in this implementation a computational graph can only be used *once*. This is sufficient and simple to use in many applications since a user does not need to think about allocating and freeing storage. However, in more complex applications it might rather be desirable to keep the graph for further computations. Then the graph generation and the computation of the function value must be separated into two phases, i.e. first the graph must be constructed, second the function values can be computed (forward sweep) and third the adjoint values can be computed (reverse sweep). In such a setting the second and third step can be repeated as often as desired without reconstructing the graph each time.

The last function, `eval_grad`, in this module can be used to get an easy access to the function value *and* the gradient of an expression: the first argument is the expression to be evaluated, and the second argument is the vector of the n independent variables. Then a call of `eval_grad` returns a vector of length $n + 1$ which contains the function value in the first component and the gradient in the remaining n components.

```
MODULE igradrev;

USE i_ari;

{----------------------------------------------------------------------------}
{ Computation of gradients for real functions using the reverse mode         }
{ of automatic differentiation.                                              }
{                                                                            }
{ This module contains type definitions, operators and functions which       }
{ enable the user to build a tree as internal representation for a function  }
{ f(x1,...xn) of n variable, to compute the value of that function for       }
{ given values of x1,...,xn and to compute the gradient of f for these       }
{ values of x1,...,xn using the reverse mode of automatic differentiation.   }
{                                                                            }
{ The type  o p e r a t i o n  is used to mark each node of the tree with    }
{ the type of operation which it represents.                                 }
{                                                                            }
{ The record-type  f n o d e  contains all informations which have to be     }
{ stored in each node ( = partial formula ):                                 }
{      op   -  the operation which has to be done in this node               }
{      sub  -  a counter which counts the number of father-nodes pointing    }
{              to the actual node. This counter is initialized with -1       }
{              when the node is created ( in function newf ) and increased   }
{              by 1 each time a new pointer is set to this node by one of    }
{              the operators and functions defined here. Thus, sub should    }
{              always be >=0 ( except for the root of the tree and certain   }
{              constants generated by function con ).                        }
{              The value sub+1 therefore counts how often the subtree        }
{              starting at this node occurs as a partial expression in f.    }
{              This value is incremented by 1 during tree construction time. }
{              When the reverse sweep of automatic differentiation is in-    }
{              voked by a function call to eval, then sub is decremented by  }
{              1 each time the recursive function rev (local to eval) enters }
{              a node. The reverse sweep is done for a node only if sub=0.   }
{              This ensures that the computation of the "adjoint value" g    }
{              has been completed for this node and that it can be used now  }
{              for further accumulation in its subexpressions.               }
{              Therefore during the reverse sweep the computation for each   }
{              subexpression occurring several times in f is done only once. }
{      l,r  -  pointers to the left and right operands of the node.          }
{              In the case of unary operators of function only l is used.    }
{      x    -  interval value which contains the function value of the       }
{              partial expression represented by the current node. These     }
{              values are computed simultanuously with the tree construc-    }
{              tion. Therefore during tree construction the computation for  }
{              each subexpression occurring several times in f is done only  }
{              once since for an already existing subtree its value x is     }
{              readily available in its root.                                }
{      g    -  "adjoint values" which are computed during the reverse sweep  }
{              of reverse automatic differentiation. They contain the        }
{              partial derivative of f with respect to the current node      }
{              viewed as independent variable.                               }
```

```
{              In the nodes for the independent variables x1,...,xn,        }
{              the corresponding values g1,...,gn are just the components   }
{              of the gradient of f, after the reverse sweep is completed.  }
{                                                                           }
{ The type  f o r m u l a  is used to construct the binary tree with the    }
{ operators and functions contained in this module.                         }
{---------------------------------------------------------------------------}


GLOBAL TYPE

   operation      =  GLOBAL
                    ( op_const,  op_variable, op_uminus,
                      op_plus,   op_minus,    op_mult,   op_div,
                      op_sqr,    op_sqrt,     op_power,
                      op_exp,    op_exp2,     op_exp10,
                      op_ln,     op_log2,     op_log10,
                      op_sin,    op_cos,      op_tan,    op_cot,
                      op_arcsin, op_arccos,   op_arctan, op_arccot,
                      op_sinh,   op_cosh,     op_tanh,   op_coth,
                      op_arsinh, op_arcosh,   op_artanh, op_arcoth    );

   formula        = GLOBAL ↑formula_node;

   formula_vector = GLOBAL DYNAMIC ARRAY [*] OF formula;

   formula_node   = GLOBAL RECORD
                       op  : operation;
                       sub : INTEGER;
                       x,g : INTERVAL;
                       l,r : formula;
                    END;

{---------------------------------------------------------------------------}
{ unique function for generation of formula nodes                           }
{---------------------------------------------------------------------------}

FUNCTION newf( op : operation; a,b : formula; x : INTERVAL ) : formula;
VAR f : formula;
BEGIN
   NEW( f );
   f↑.op  := op;
   f↑.l   := a;
   f↑.r   := b;
   f↑.x   := x;
   f↑.g   := 0.0;
   f↑.sub:= -1;
   newf   := f;
END;


{---------------------------------------------------------------------------}
{ functions for generation of variables- and const-nodes                    }
{---------------------------------------------------------------------------}

GLOBAL FUNCTION con ( x : INTERVAL  ) : formula;
BEGIN
   con:= newf( op_const, NIL, NIL, x );
END;

GLOBAL FUNCTION con ( x : REAL  ) : formula;
BEGIN
   con:= newf( op_const, NIL, NIL, INTVAL(x) );
END;
```

```
GLOBAL FUNCTION variable( x : INTERVAL ) : formula;
BEGIN
   variable:= newf( op_variable, NIL, NIL, x );
END;

{-----------------------------------------------------------------------------}
{ Monadic operators + and - for formula operands                              }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( a : formula ) f_uplus : formula;
{ no new node generated here ! }
BEGIN
   f_uplus:= a;
END;

GLOBAL OPERATOR - ( a : formula ) f_umin : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   f_umin:= newf( op_uminus, a, NIL, -a↑.x );
END;

{-----------------------------------------------------------------------------}
{ Operators +, -, * and / for two formula operands                            }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( a,b : formula ) ff_add : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   b↑.sub:= b↑.sub + 1;
   ff_add:= newf( op_plus, a, b, a↑.x + b↑.x );
END;

GLOBAL OPERATOR - ( a,b : formula ) ff_sub : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   b↑.sub:= b↑.sub + 1;
   ff_sub:= newf( op_minus, a, b, a↑.x - b↑.x );
END;

GLOBAL OPERATOR * ( a,b : formula ) ff_mul : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   b↑.sub:= b↑.sub + 1;
   ff_mul:= newf( op_mult, a, b, a↑.x * b↑.x );
END;

GLOBAL OPERATOR / ( a,b : formula ) ff_div : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   b↑.sub:= b↑.sub + 1;
   ff_div:= newf( op_div, a, b, a↑.x / b↑.x );
END;

{-----------------------------------------------------------------------------}
{ Operators +, -, * and / for one formula and one INTERVAL operand            }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( a : formula; b : INTERVAL ) fi_add : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   fi_add:= newf( op_plus, a, con(b), a↑.x + b );
END;
```

```
GLOBAL OPERATOR - ( a : formula; b : INTERVAL ) fi_sub : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   fi_sub:= newf( op_minus, a, con(b), a↑.x - b );
END;

GLOBAL OPERATOR * ( a : formula; b : INTERVAL ) fi_mul : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   fi_mul:= newf( op_mult, a, con(b), a↑.x * b );
END;

GLOBAL OPERATOR / ( a : formula; b : INTERVAL ) fi_div : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   fi_div:= newf( op_div, a, con(b), a↑.x / b );
END;

{----------------------------------------------------------------------------}
{ Operators +, -, * and / for one INTERVAL and one formula operand           }
{----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( a : INTERVAL; b : formula ) if_add : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   if_add:= newf( op_plus, con(a), b, a + b↑.x );
END;

GLOBAL OPERATOR - ( a : INTERVAL; b : formula ) if_sub : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   if_sub:= newf( op_minus, con(a), b, a - b↑.x );
END;

GLOBAL OPERATOR * ( a : INTERVAL; b : formula ) if_mul : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   if_mul:= newf( op_mult, con(a), b, a * b↑.x );
END;

GLOBAL OPERATOR / ( a : INTERVAL; b : formula ) if_div : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   if_div:= newf( op_div, con(a), b, a / b↑.x );
END;

{----------------------------------------------------------------------------}
{ Operators +, -, * and / for one formula and one REAl operand               }
{----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( a : formula; b : REAL ) fr_add : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   fr_add:= newf( op_plus, a, con(b), a↑.x + b );
END;

GLOBAL OPERATOR - ( a : formula; b : REAL ) fr_sub : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   fr_sub:= newf( op_minus, a, con(b), a↑.x - b );
END;

GLOBAL OPERATOR * ( a : formula; b : REAL ) fr_mul : formula;
BEGIN
```

```
   a↑.sub:= a↑.sub + 1;
   fr_mul:= newf( op_mult, a, con(b), a↑.x * b );
END;

GLOBAL OPERATOR / ( a : formula; b : REAL ) fr_div : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   fr_div:= newf( op_div, a, con(b), a↑.x / b );
END;

{----------------------------------------------------------------------------}
{ Operators +, -, * and / for one REAL and one formula operand               }
{----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( a : REAL; b : formula ) rf_add : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   rf_add:= newf( op_plus, con(a), b, a + b↑.x );
END;

GLOBAL OPERATOR - ( a : REAL; b : formula ) rf_sub : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   rf_sub:= newf( op_minus, con(a), b, a - b↑.x );
END;

GLOBAL OPERATOR * ( a : REAL; b : formula ) rf_mul : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   rf_mul:= newf( op_mult, con(a), b, a * b↑.x );
END;

GLOBAL OPERATOR / ( a : REAL; b : formula ) rf_div : formula;
BEGIN
   b↑.sub:= b↑.sub + 1;
   rf_div:= newf( op_div, con(a), b, a / b↑.x );
END;

{----------------------------------------------------------------------------}
{      elementary functions for formula                                      }
{----------------------------------------------------------------------------}

GLOBAL FUNCTION sqr ( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   sqr:= newf( op_sqr, a, NIL, SQR(a↑.x) );
END;

GLOBAL FUNCTION sqrt( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   sqrt:= newf( op_sqrt, a, NIL, SQRT(a↑.x) );
END;

GLOBAL FUNCTION power( a : formula; b : INTERVAL ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   power:= newf( op_power, a, con(b), POWER(a↑.x,b) );
END;

GLOBAL FUNCTION power( a : formula; b : REAL ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   power:= newf( op_power, a, con(b), POWER(a↑.x,INTVAL(b)) );
```

```
END;

GLOBAL FUNCTION power( a,b : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   b↑.sub:= b↑.sub + 1;
   power:= newf( op_power, a, b, POWER(a↑.x,b↑.x) );
END;

GLOBAL FUNCTION exp( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   exp:= newf( op_exp, a, NIL, EXP(a↑.x) );
END;

GLOBAL FUNCTION exp2( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   exp2:= newf( op_exp2, a, NIL, EXP2(a↑.x) );
END;

GLOBAL FUNCTION exp10( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   exp10:= newf( op_exp10, a, NIL, EXP10(a↑.x) );
END;

GLOBAL FUNCTION ln( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   ln:= newf( op_ln, a, NIL, LN(a↑.x) );
END;

GLOBAL FUNCTION log2( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   log2:= newf( op_log2, a, NIL, LOG2(a↑.x) );
END;

GLOBAL FUNCTION log10( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   log10:= newf( op_log10, a, NIL, LOG10(a↑.x) );
END;

GLOBAL FUNCTION sin( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   sin:= newf( op_sin, a, NIL, SIN(a↑.x) );
END;

GLOBAL FUNCTION cos( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   cos:= newf( op_cos, a, NIL, COS(a↑.x) );
END;

GLOBAL FUNCTION tan( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   tan:= newf( op_tan, a, NIL, TAN(a↑.x) );
END;

GLOBAL FUNCTION cot( a : formula ) : formula;
```

```
BEGIN
   a↑.sub:= a↑.sub + 1;
   cot:= newf( op_cot, a, NIL, COT(a↑.x) );
END;

GLOBAL FUNCTION arcsin( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   arcsin:= newf( op_arcsin, a, NIL, ARCSIN(a↑.x) );
END;

GLOBAL FUNCTION arccos( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   arccos:= newf( op_arccos, a, NIL, ARCCOS(a↑.x) );
END;

GLOBAL FUNCTION arctan( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   arctan:= newf( op_arctan, a, NIL, ARCTAN(a↑.x) );
END;

GLOBAL FUNCTION arccot( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   arccot:= newf( op_arccot, a, NIL, ARCCOT(a↑.x) );
END;

GLOBAL FUNCTION sinh( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   sinh:= newf( op_sinh, a, NIL, SINH(a↑.x) );
END;

GLOBAL FUNCTION cosh( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   cosh:= newf( op_cosh, a, NIL, COSH(a↑.x) );
END;

GLOBAL FUNCTION tanh( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   tanh:= newf( op_tanh, a, NIL, TANH(a↑.x) );
END;

GLOBAL FUNCTION coth( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   coth:= newf( op_coth, a, NIL, COTH(a↑.x) );
END;

GLOBAL FUNCTION arsinh( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   arsinh:= newf( op_arsinh, a, NIL, ARSINH(a↑.x) );
END;

GLOBAL FUNCTION arcosh( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   arcosh:= newf( op_arcosh, a, NIL, ARCOSH(a↑.x) );
END;
```

```
GLOBAL FUNCTION artanh( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   artanh:= newf( op_artanh, a, NIL, ARTANH(a↑.x) );
END;

GLOBAL FUNCTION arcoth( a : formula ) : formula;
BEGIN
   a↑.sub:= a↑.sub + 1;
   arcoth:= newf( op_arcoth, a, NIL, ARCOTH(a↑.x) );
END;

{----------------------------------------------------------------------------}
{   Function for evaluation of expression f ( i.e. read out its value f↑.x   }
{   which was already computed during tree-connstruction of f ).             }
{   Furthermore, start reverse sweep through tree for f to compute the       }
{   adjoint values g in each node.                                           }
{   All variables and constants in the tree must be initialized properly !   }
{----------------------------------------------------------------------------}

GLOBAL FUNCTION eval( f : formula ) : INTERVAL;

procedure reverse_sweep( f : formula );
BEGIN
   WITH f↑ DO
   BEGIN
      CASE op OF
         op_const    : ;
         op_variable: ;
         op_uminus   : l↑.g:= l↑.g - g;
         op_plus     : BEGIN
                          l↑.g:= l↑.g + g;
                          r↑.g:= r↑.g + g;
                       END;
         op_minus    : BEGIN
                          l↑.g:= l↑.g + g;
                          r↑.g:= r↑.g - g;
                       END;
         op_mult     : BEGIN
                          l↑.g:= l↑.g + g*r↑.x;
                          r↑.g:= r↑.g + g*l↑.x;
                       END;
         op_div      : BEGIN
                          l↑.g:= l↑.g + g/r↑.x;
                          r↑.g:= r↑.g - g*x/r↑.x;
                       END;
         op_sqr      : l↑.g:= l↑.g + g*2.0*l↑.x;
         op_sqrt     : l↑.g:= l↑.g + g*0.5/x;
         op_power    : BEGIN
                          IF r↑.op <> op_const THEN
                               r↑.g:= r↑.g + g*x*LN(l↑.x);
                          IF l↑.op <> op_const THEN
                               l↑.g:= l↑.g + g*r↑.x*POWER(l↑.x,r↑.x-1);
                       END;
         op_exp      : l↑.g:= l↑.g + g*x;
         op_exp2     : l↑.g:= l↑.g + g*x*LN(INTVAL(2.0));
         op_exp10    : l↑.g:= l↑.g + g*x*LN(INTVAL(10.0));
         op_ln       : l↑.g:= l↑.g + g/l↑.x;
         op_log2     : l↑.g:= l↑.g + g/(l↑.x*LN(INTVAL(2.0)));
         op_log10    : l↑.g:= l↑.g + g/(l↑.x*LN(INTVAL(10.0)));
         op_sin      : l↑.g:= l↑.g + g*cos(l↑.x);
         op_cos      : l↑.g:= l↑.g - g*sin(l↑.x);
         op_tan      : l↑.g:= l↑.g + g*(sqr(x)+1.0);
```

```
         op_cot     : l↑.g:= l↑.g - g*(sqr(x)+1.0);
         op_arcsin  : l↑.g:= l↑.g + g/sqrt(1.0-sqr(l↑.x));
         op_arccos  : l↑.g:= l↑.g - g/sqrt(1.0-sqr(l↑.x));
         op_arctan  : l↑.g:= l↑.g + g/(1.0+sqr(l↑.x));
         op_arccot  : l↑.g:= l↑.g - g/(1.0+sqr(l↑.x));
         op_sinh    : l↑.g:= l↑.g + g*cosh(l↑.x);
         op_cosh    : l↑.g:= l↑.g + g*sinh(l↑.x);
         op_tanh    : l↑.g:= l↑.g + g*(1.0-sqr(x));
         op_coth    : l↑.g:= l↑.g + g*(1.0-sqr(x));
         op_arsinh  : l↑.g:= l↑.g + g/sqrt(sqr(l↑.x)+1.0);
         op_arcosh  : l↑.g:= l↑.g + g/sqrt(sqr(l↑.x)-1.0);
         op_artanh  : l↑.g:= l↑.g + g/(1.0-sqr(l↑.x));
         op_arcoth  : l↑.g:= l↑.g + g/(1.0-sqr(l↑.x));
      END;
      IF l <> NIL THEN IF l↑.sub<=0 THEN reverse_sweep(l)
                                    ELSE l↑.sub:= l↑.sub - 1;
      IF r <> NIL THEN IF r↑.sub<=0 THEN reverse_sweep(r)
                                    ELSE r↑.sub:= r↑.sub - 1;
   END;
   IF f↑.op <> op_variable THEN DISPOSE(f); { dispose nodes unless variables }
END;

BEGIN
   eval:= f↑.x;
   f↑.g:= 1;
   reverse_sweep( f );
END;

{----------------------------------------------------------------------------}
{   Function for evaluation of an expression f and its gradient f_x          }
{   with the values contained in the vector of variables x.                  }
{   The adjoint values x[i]↑.g in x are reset, the values x[i]↑.x of the     }
{   variables must be initialized properly.                                  }
{   The result is returned in an IVECTOR with the function value f(x)        }
{   having index LB(x)-1 and the gradient indices LB(x)..UB(x).              }
{----------------------------------------------------------------------------}

GLOBAL FUNCTION eval_grad( f : formula; x : formula_vector )
                                                       : IVECTOR[LB(x)-
1..UB(x)];
VAR i : INTEGER;
BEGIN
   FOR i:= LB(x) TO UB(x) DO x[i]↑.g:= 0.0;     { reset adjoint values      }
                                                 { all x[i] must be variables }
   eval_grad[LB(x)-1]:= eval( f );              { return function value in  }
                                                 { component LB(x)-1         }
   FOR i:= LB(x) TO UB(x) DO eval_grad[i]:= x[i]↑.g; { return gradient in    }
                                                    { components LB(x)..UB(x)}
END;

END.
```

Concluding this section we mention as in the case of the itaylor module that we can convert the module by only a few simple changes into a module rgradrev for the computation of gradients using the REAL data type, or to a module cgradrev using the COMPLEX data type or even to a module cigradrev using the CINTERVAL data type (i.e. complex intervals).

Apart from renaming the module the following changes should be made:

- change the `USE`-clause; no `USE` is necessary for REAL, `USE c_ari` is needed for COMPLEX and `USE i_ari,c_ari,ci_ari` are needed for CINTERVAL.

- change all occurences of the data type INTERVAL to REAL, COMPLEX or CINTERVAL as desired. The name of the data type formula needs no change as long as the modified module is used alone. If it is used together with a second one (interval and cinterval, say) then a suitable change of this name should be considered.
- discard the calls to the intrinsic function INTVAL in the cases of REAL and COMPLEX.
- operators with mixed operand types (formula, REAL, INTERVAL, COMPLEX and/or CINTERVAL) may be added or deleted as desired.

There are however versions of PASCAL-XSC where many complex standard functions are not implemented (and also not those for complex intervals). With these versions only the basic arithmetic will work in complex, the standard functions will not.

5.2.5 Test Results

In the example which we present here for the use of the module igradrev we write a simple verifying solver for a system of nonlinear equations $f(x) = 0$. The module is used to generate an enclosure for the Jacobian matrix $f'(x)$ i.e. we compute each row $f_i'(x)$ of $f'(x)$ by computing the gradient of $f_i(x)$ with the operators and functions in the module igradrev.

We want to find a root of the nonlinear system of equations

$$f(x) = 0, \quad f : \mathbb{R}^n \to \mathbb{R}^n \tag{5.31}$$

i.e. in our verifying context we want to proof that there exists a solution x^* of (5.31) and compute an enclosure thereof.

Starting from an approximation $\tilde{x}$ we compute iterates with the so called Krawczyk operator K

$$\begin{aligned} &[x_{k+1}] := K([x_k]) := x_m - Rf(x_m) + (I - Rf'([x_k]))([x_k] - x_m), \\ &\quad x_m \in [x_k] \text{ e.g. } \; x_m = \text{mid}([x_k]) \end{aligned} \tag{5.32}$$

which is a Newton like iteration. Here R is an approximate inverse for the Jacobian matrix at some point in $[x_k]$. Usually one takes $f'(x_m)$, x_m as in (5.32) and computes an approximate inverse of this matrix.

It can be shown that there exists a solution $x^* \in [x_k]$ if the operator maps $[x_k]$ into itself i.e. if the *existence test*

$$[x_{k+1}] = K([x_k]) \subseteq [x_k] \tag{5.33}$$

holds true, see e.g. . [284], [248].

In order to accelerate convergence in the sense that (5.33) will be satisfied as soon as possible we also introduce an ϵ-*inflation*

$$\text{blow}([u], \epsilon) := (1 + \epsilon)[u] - \epsilon[u]$$

which increases the diameter of $[u]$ by a factor of 2ϵ and leaves its midpoint fixed. If $\mathrm{diam}([u]) = 0$ then we set $\mathrm{blow}([u], \epsilon)$ to some interval with midpoint $u = [u]$ and a sufficiently small diameter, e.g. on a computer we take the interval of the next lower and the next upper floating-point number. A solution lying close to $[u]$ is thus more likely to be contained in $\mathrm{blow}([u], \epsilon)$, see [284].

Thus, the algorithm which is implemented in our small demonstration program is as follows:

$X_1 := \tilde{x}$ (given)
$k := 0$
repeat
  $k := k + 1$
  $X_0 := \mathrm{blow}(X_1, \epsilon)$
  $x_m := \mathrm{mid}(X_0)$
  $X_1 := x_m - Rf(x_m) + (I - Rf'(X_0))(X_0 - x_m)$
until $X_1 \subset X_0$ **or** $k = k_{max}$

where $\tilde{x}$ is a given approximate solution, ϵ and k_{max} are suitably chosen constants and $f'(X_0)$ is computed row-wise by automatic differentiation with the function eval_grad in module igradrev. The approximate inverse R is computed with the function `MINV` from the standard PASCAL-XSC module lss_aprx.

As a sample function f we choose a simple two-dimensional system which consists of two circles with two intersection points at $(0.5, \pm\sqrt{3}/2)$:

$$\begin{aligned} x_1^2 + x_2^2 - 1 &= 0 \\ (x_1^2 - 1) + x_2^2 - 1 &= 0 \end{aligned}$$

The program uses the module igradrev, some standard arithmetic modules and the module lss_aprx, which contains a function `MINV` for the computation of an approximate inverse $R \approx A^{-1}$ of a matrix A. This module is also a standard module of PASCAL-XSC.

```
PROGRAM ignlss;

USE i_ari,mv_ari,mvi_ari,lss_aprx,igradrev;

CONST n = 2;

FUNCTION f( y : IVECTOR ) : IMATRIX[1..n,0..n];
VAR x : formula_vector[1..n];
    s   : formula;
    i,j : INTEGER;
BEGIN
   FOR i:= 1 TO n DO x[i]:= variable( y[i] );        { define x as variable  }
   { evaluate f_1, f_2 and their gradients    }
   f[1]:= eval_grad( sqr(  x[1]  ) + sqr(x[2]) - 1.0, x );
   f[2]:= eval_grad( sqr(x[1]-1.0) + sqr(x[2]) - 1.0, x );
END;

FUNCTION newton_step( x0 : IVECTOR ) : IVECTOR[1..n];
```

```
{ computes one step with the Newton like Krawczyk operator }
VAR f0,fm : IVECTOR[1..n];
    xm    : RVECTOR[1..n];
    df0   : IMATRIX[1..n,1..n];
    df1   : RMATRIX[1..n,1..n];
    r_x   : IMATRIX[1..n,0..n];
    error : INTEGER;
BEGIN
   r_x:= f(x0);                      { compute f and f' over interval x0 }
   f0 := r_x[*,0];                   { extract function value f into f0  }
   df0:= r_x[*,1..n];                { extract Jacobian f' intp df0      }
   df1:= MID(df0);                   { compute approximate inverse of    }
   MINV( df1, error );               {    midpoint-matrix of Jacobian f' }
   IF error<>0 THEN WRITELN('ERROR in MINV: matrix ill conditioned !!');
   xm := MID(x0);
   r_x:= f(INTVAL(xm));              { compute f for midpoint x_m of x_0 }
   fm := r_x[*,0];                   { (ignore Jacobian here)            }
   newton_step:= ##(xm - df1*fm) + ##( ID(df0) - df1*df0 )*(x0-xm);
END;

CONST epsilon = 0.1;
VAR x0,x1    : IVECTOR[1..n];
    k,k_max : INTEGER;
BEGIN
   WRITE('starting vector           : ');  READ( x1 );
   WRITE('max. number of iterations : ');  READ(k_max);
   k:= 0;
   REPEAT
      k:= k + 1;
      x0:= BLOW( x1, epsilon );
      x1:= newton_step( x0 );
      WRITE( 'k = ', k:3, x1 );
      IF x1 IN x0 THEN WRITE('--> Inclusion ! <--');
   UNTIL ( x1 IN x0 ) OR ( k = k_max );
END.
```

A sample output of the program is

```
starting vector           : 0.2 0.3
max. number of iterations : 10
k =   1
[  4.999999999999997E-001,  5.000000000000003E-001 ]
[  1.549999999999999E+000,  1.550000000000001E+000 ]
k =   2
[   4.99999999999999E-001,   5.00000000000001E-001 ]
[  1.016935483870967E+000,  1.016935483870969E+000 ]
k =   3
[  4.999999999999997E-001,  5.000000000000005E-001 ]
[  8.772226983193059E-001,  8.772226983193066E-001 ]
k =   4
[  4.999999999999996E-001,  5.000000000000004E-001 ]
[  8.660968676243165E-001,  8.660968676243173E-001 ]
k =   5
[  4.999999999999996E-001,  5.000000000000004E-001 ]
[  8.660254067327692E-001,  8.660254067327699E-001 ]
k =   6
[  4.999999999999998E-001,  5.000000000000004E-001 ]
[  8.660254037844383E-001,  8.660254037844389E-001 ]
k =   7
[  4.999999999999997E-001,  5.000000000000004E-001 ]
[  8.660254037844383E-001,  8.660254037844389E-001 ]
--> Inclusion ! <--
```

and for the second solution

```
starting vector           : 0.6 -0.7
max. number of iterations : 10
k =   1
[  4.999999999999997E-001,  5.000000000000003E-001 ]
[ -8.928571428571432E-001, -8.928571428571425E-001 ]
k =   2
[  4.999999999999997E-001,  5.000000000000003E-001 ]
[ -8.664285714285717E-001, -8.664285714285712E-001 ]
k =   3
[  4.999999999999997E-001,  5.000000000000003E-001 ]
[ -8.660254975856792E-001, -8.660254975856787E-001 ]
k =   4
[  4.999999999999998E-001,  5.000000000000004E-001 ]
[ -8.660254037844440E-001, -8.660254037844434E-001 ]
k =   5
[  4.999999999999998E-001,  5.000000000000004E-001 ]
[ -8.660254037844389E-001, -8.660254037844383E-001 ]
k =   6
[  4.999999999999997E-001,  5.000000000000004E-001 ]
[ -8.660254037844389E-001, -8.660254037844383E-001 ]
--> Inclusion ! <--
```

5.2.6 Notes and References

The module can easily be modified to the use of different data types as has already been discussed at the end of Section 5.2.4.

Another possible modification has also been indicated at the beginning of Section 5.2.4: in many applications it is more desirable to keep the graph in memory after the evaluation of forward and reverse sweep in order to do multiple evaluations of the same expression for different arguments. Then the allocation and deallocation of memory is more in the responsibility of the user of the module who has to take more care about this.

Instead of a computational graph we also could use a linear code list like e.g. a sequence of simple instructions in some programming language or a somehow coded sequence of such instructions stored in an array. Then this code list must be interpreted in the forward and reverse sweep and all relevant data must be stored in another list which may be in main memory and/or on a different storage device. There may be two advantaged to such an approach, first there is no need to use pointers which might sometimes make inefficient use of memory (by varying over wide address ranges) and second a code list does not necessarily need to stay in memory completely for all the time, so its size can be larger than the available main memory. The same holds true also for the generated intermediate values x_i and g_i which may be stored elsewhere in this case. See e.g. Griewank's implementation using a so-called 'tape' for these data.

Yet another possibility is a compiler technical approach where we take the function f as a PASCAL-XSC or FORTAN function, say, and write a program which reads this function and compiles it to another function in the same programming language by adding instructions which compute the desired derivatives. For such an approach see e.g. [209], [39] and others.

In [116] Griewank has presented a technique which may reduce the storage complexity of reverse automatic differentiation logarithmically while increasing the time complexity somewhat. This is achieved by taking repeated 'snapshots' of the complete state of the computation and storing these snapshots only during the forward sweep instead of storing all intermediate values. In the reverse sweep then all missing values can be recomputed by use of the values taken from the snapshots.

Further reduction of space and time complexity of reverse automatic differentiation can be achieved by introducing basic operations on higher levels such as matrix-vector operations or even considering problem solvers as basic operations of a function. See e.g. [301] and [302].

Also generalizations to higher derivatives are possible. There exist reverse methods for the computation of Hessian matrices etc. Similarly special methods for the product of a Hessian times a vector and for similar special cases have been developed, see Griewank and others.

5.3 Functions in Several Variables, Fast Directional Derivatives

5.3.1 Introduction

In many methods and applications it is often not the gradient ∇f of a function f itself but rather the scalar product $\nabla f \cdot q$ with a certain fixed vector q which is needed. This occurs e.g. in many optimization methods where directional derivatives are needed.

This special problem is treated in this section. From the previous Section 5.2 we can trivially derive a method to solve this task with a cost that is only a constant times the cost for computing f alone, since we can get the gradient ∇f for this cost and assuming that the cost for evaluating f increases at least linearly with the dimension n the additional scalar product $\nabla f \cdot q$ does not influence the final cost ratio. However, we saw that computing the gradient with the reverse mode of automatic differentiation may need a lot of memory for all intermediate results.

In this section we present a forward method for the computation of $\nabla f \cdot q$ which does not suffer from this high need for memory and which nevertheless requires also an overhead which is less than a constant multiple of the cost for evaluating f.

This forward method is due to H. Fischer, [100], and was extended to standard functions by H.-C. Fischer, [101], [104].

5.3.2 Theoretical Background

In the following we will use the abbreviation

$$f_q := \nabla f \cdot q$$

for the scalar product of the gradient of a function $f = f(x_1, \ldots, x_n)$ with the fixed vector $q = (q_1, \ldots, q_n)$. Just as in the case of Taylor coefficients, Section 5.1, we consider the derivative, which is a gradient now, for each basic operation in f.

For the trivial cases of a constant c or an independent variable x_i and for the arithmetic operations (with u, v functions of x) we see immediately:

$$\begin{array}{rclcrcl} w & = & c & : & w_q & = & 0 \\ w & = & x_i & : & w_q & = & q_i \\ w & = & u \pm v & : & w_q & = & u_q \pm v_q \\ w & = & u \cdot v & : & w_q & = & uv_q + u_q v \\ w & = & u/v & : & w_q & = & (u_q - wv_q)/v \end{array} \tag{5.34}$$

For an arbitrary differentiable function $w(u)$ we get just as easily

$$\nabla w(u) = w'(u) \cdot \nabla u \quad \text{i.e.} \quad (w(u))_q = w'(u)u_q \ . \tag{5.35}$$

For the standard functions w in PASCAL-XSC these derivatives are listed e.g. in (5.30).

As we see from the previous two equations (5.34), (5.35) we only need to compute two scalar quantities: f and f_q using the above relations recursively. The additional overhead for the computation of f_q is obviously bounded by the number of operations which are needed additionally. From (5.34) we see that for $\pm$ this is the same number for w_q as it is for w. For $\cdot$ and $/$ it is three times as much for w_q as it is for w. Similarly for standard functions $w(u)$ the number of operations needed to compute $(w(u))_q$ is only a small multiple from the cost of computing $w(u)$, compare e.g. (5.30).

5.3.3 Algorithms

As in the case of the Taylor coefficients we implement our forward method for the computation of f and $f_q = \nabla f \cdot q$ as an arithmetic operating on pairs of numbers in this case. Introducing the pairs $W = (w, w_q)$ we can write down very simple operators and functions performing the operations to compute f in the first component and to compute f_q in the second component of the pairs.

For these pairs we use the PASCAL-XSC data type

```
TYPE  igrad_q  = RECORD f,fq : INTERVAL END;
```

The algorithms are so simple and easy to implement that we list only a few examples. The rest can be found in the program listing.

Algorithm 5.18: $+(u, v)$ {operator}

{Computation of $u + v$ and $\nabla(u + v) \cdot q$}
input: *igrad_q* u, v
output: Enclosure of $u + v$ and $\nabla(u + v) \cdot q$ (operator result)

$w.f := u.f + v.f$
$w.fq := u.fq + v.fq$
return $w.f, w.fq$

In the multiplication operator as well as in the division operator we make use of an exact scalar product:

Algorithm 5.19: $\cdot\,(u, v)$ {operator}

{Computation of $u \cdot v$ and $\nabla(u \cdot v) \cdot q$}
input: *igrad_q* u, v
output: Enclosure of $u \cdot v$ and $\nabla(u \cdot v) \cdot q$ (operator result)

$w.f := u.f \cdot v.f$
$w.fq := \Diamond(u.fq \cdot v.f + u.f \cdot v.fq)$
return $w.f, w.fq$

Finally, as an example for the standard functions we list the algorithm for arsinh:

Algorithm 5.20: Arsinh (u) {function}

{Computation of $\operatorname{arsinh}(u)$ and $\nabla(\operatorname{arsinh}(u)) \cdot q$}
input: *igrad_q* u
output: enclosure of $\operatorname{arsinh}(u)$ and $\nabla(\operatorname{arsinh}(u)) \cdot q$ (function result)

$w.f :=$arsinh$(u.f)$
w.fq:= u.fq / $\sqrt{\text{u.f}^2 + 1}$
Arsinh.f:= $w.f$
Arsinh.fq:= $w.fq$

5.3.4 PASCAL-XSC Program Code

In addition to the operators and functions discussed in the previous Section 5.3.3 the following module `igq_ari` contains conversion operators and functions for converting constants to type igrad_q (by assignments) and to convert two component of two vectors to type igrad_q where one of the vectors must contain the independent variables x and the other the fixed vector q.

Also an additional type igq_vector is introduced in this module which is a vector of component type igrad_q, plus two conversions from rvector and from ivector to igq_vector as a vector of constants (implemented as overloaded assignments) and two conversions from rvector and from ivector to igq_vector as a vector of independent variables (implemented as functions).

```
MODULE igq_ari;

{-----------------------------------------------------------------------}
{  Module for the computation of functions and the product of their    }
{  gradient with a fixed vector q by use of automatic differentiation   }
```

```
{   ( in the forward mode ).                                                 }
{----------------------------------------------------------------------------}

USE i_ari;

GLOBAL TYPE igrad_q    = GLOBAL RECORD f,fq : INTERVAL END;
            igq_vector = GLOBAL DYNAMIC ARRAY [*] OF igrad_q;

                             {  In a record igrad_q            }
                             {  f contains the function value, }
                             {  and fq contains the scalar     }
                             {  product of the gradient of f   }
                             {  with the fixed vector q.       }

{----------------------------------------------------------------------------}
{   Conversion functions and operators from REAL and INTERVAL                }
{   to igrad_q.  ( generation of igrad_q for constants )                     }
{----------------------------------------------------------------------------}


GLOBAL OPERATOR := ( VAR g : igrad_q; c : REAL );
{ initializes g.f with c and g.fq with zero                 }
{ i.e. generates an igrad_q corresponding to a constant     }
BEGIN
   g.f  := c;
   g.fq:= 0.0;
END;

GLOBAL OPERATOR := ( VAR g : igrad_q; c : INTERVAL );
{ initializes g.f with c and g.fq with zero                 }
{ i.e. generates an igrad_q corresponding to a constant     }
BEGIN
   g.f  := c;
   g.fq:= 0.0;
END;

GLOBAL OPERATOR := ( VAR g : igq_vector; c : RVECTOR );
{ initializes all components g[k].f with the components of  }
{ c and g[k].fq with zero, i.e. generates a vector of       }
{ constants.                                                }
VAR k : INTEGER;
BEGIN
   FOR k:= LB(g) TO UB(g) DO BEGIN
                                g[k].f  := c[k];
                                g[k].fq:= 0.0;
                             END;
END;

GLOBAL OPERATOR := ( VAR g : igq_vector; c : IVECTOR );
{ initializes all components g[k].f with the components of  }
{ c and g[k].fq with zero, i.e. generates a vector of       }
{ constants.                                                }
VAR k : INTEGER;
BEGIN
   FOR k:= LB(g) TO UB(g) DO BEGIN
                                g[k].f  := c[k];
                                g[k].fq:= 0.0;
                             END;
END;

{----------------------------------------------------------------------------}
{   Conversion functions for generating an igrad_q for an independent        }
{   variable x[k], i.e.  grad( x[k]*q ) = q[k] or for a vector x of          }
{   independent variables.                                                   }
```

```
{-----------------------------------------------------------------------------}

GLOBAL FUNCTION igradq( VAR x,q : RVECTOR ) : igq_vector[LB(x)..UB(x)];
{ initializes the resulting igrad_q with x[k] and with q[k] i.e.        }
{ generates an igrad_q corresponding to the independent variable x[k]. }
VAR k : INTEGER;
BEGIN
   FOR k:= LB(x) TO UB(x) DO BEGIN
                                igradq[k].f := x[k];
                                igradq[k].fq:= q[k];
                             END;
END;

GLOBAL FUNCTION igradq( VAR x,q : IVECTOR ) : igq_vector[LB(x)..UB(x)];
{ initializes the resulting igrad_q with x[k] and with q[k] i.e.        }
{ generates an igrad_q corresponding to the independent variable x[k]. }
VAR k : INTEGER;
BEGIN
   FOR k:= LB(x) TO UB(x) DO BEGIN
                                igradq[k].f := x[k];
                                igradq[k].fq:= q[k];
                             END;
END;

GLOBAL FUNCTION igradq( VAR x,q : RVECTOR; k : INTEGER ) : igrad_q;
{ initializes the resulting igrad_q with x[k] and with q[k] i.e.        }
{ generates an igrad_q corresponding to the independent variable x[k]. }
BEGIN
   igradq.f := x[k];
   igradq.fq:= q[k];
END;

GLOBAL FUNCTION igradq( VAR x,q : IVECTOR; k : INTEGER ) : igrad_q;
{ initializes the resulting igrad_q with x[k] and with q[k] i.e.        }
{ generates an igrad_q corresponding to the independent variable x[k]. }
BEGIN
   igradq.f := x[k];
   igradq.fq:= q[k];
END;

GLOBAL FUNCTION igradq( x,q : REAL ) : igrad_q;
{ initializes the resulting igrad_q with x and with q i.e.          }
{ generates an igrad_q corresponding to the independent variable x. }
{ x and q should be corresponding components of two vectors.        }
BEGIN
   igradq.f := x;
   igradq.fq:= q;
END;

GLOBAL FUNCTION igradq( x,q : INTERVAL ) : igrad_q;
{ initializes the resulting igrad_q with x and with q i.e.          }
{ generates an igrad_q corresponding to the independent variable x. }
{ x and q should be corresponding components of two vectors.        }
BEGIN
   igradq.f := x;
   igradq.fq:= q;
END;

{-----------------------------------------------------------------------------}
{ Monadic operators + and - for igrad_q operands                              }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u : igrad_q ) ig_uplus : igrad_q;
BEGIN
```

```
   ig_uplus:= u;
END;

GLOBAL OPERATOR - ( VAR u : igrad_q ) ig_umin : igrad_q;
BEGIN
   ig_umin.f := - u.f ;
   ig_umin.fq:= - u.fq;
END;

{-----------------------------------------------------------------------------}
{ Operators +, -, * and / for two igrad_q operands                            }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u,v : igrad_q ) igig_add : igrad_q;
BEGIN
   igig_add.f := u.f  + v.f ;
   igig_add.fq:= u.fq + v.fq;
END;

GLOBAL OPERATOR - ( VAR u,v : igrad_q ) igig_sub : igrad_q;
BEGIN
   igig_sub.f := u.f  - v.f ;
   igig_sub.fq:= u.fq - v.fq;
END;

GLOBAL OPERATOR * ( VAR u,v : igrad_q ) igig_mul : igrad_q;
BEGIN
   igig_mul.f := u.f * v.f;
   igig_mul.fq:= ##( u.fq*v.f + u.f*v.fq );
END;

GLOBAL OPERATOR / ( VAR u,v : igrad_q ) igig_div : igrad_q;
VAR h : INTERVAL;
BEGIN
            h := u.f / v.f;
   igig_div.f := h;
   igig_div.fq:= ##( u.fq - h*v.fq ) / v.f;
END;


{-----------------------------------------------------------------------------}
{ Operators +, -, * and / for one igrad_q and one REAL operand                }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u : igrad_q; v : REAL ) igr_add: igrad_q;
BEGIN
   igr_add.f := u.f + v;
   igr_add.fq:= u.fq;
END;

GLOBAL OPERATOR - ( VAR u : igrad_q; v : REAL ) igr_sub: igrad_q;
BEGIN
   igr_sub.f := u.f - v;
   igr_sub.fq:= u.fq;
END;

GLOBAL OPERATOR * ( VAR u : igrad_q; v : REAL ) igr_mul: igrad_q;
BEGIN
   igr_mul.f := u.f * v;
   igr_mul.fq:= u.fq*v;
END;

GLOBAL OPERATOR / ( VAR u : igrad_q; v : REAL ) igr_div: igrad_q;
BEGIN
```

```
   igr_div.f := u.f / v;
   igr_div.fq:= u.fq/ v;
END;

{-----------------------------------------------------------------------------}
{ Operators +, -, * and / for one REAL and one igrad_q operand                }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( u : REAL; VAR v : igrad_q ) rig_add: igrad_q;
BEGIN
   rig_add.f := u + v.f;
   rig_add.fq:= v.fq;
END;

GLOBAL OPERATOR - ( u : REAL; VAR v : igrad_q ) rig_sub: igrad_q;
BEGIN
   rig_sub.f := u - v.f;
   rig_sub.fq:= -v.fq;
END;

GLOBAL OPERATOR * ( u : REAL; VAR v : igrad_q ) rig_mul: igrad_q;
BEGIN
   rig_mul.f := u * v.f ;
   rig_mul.fq:= u*v.fq;
END;

GLOBAL OPERATOR / ( u : REAL; VAR v : igrad_q ) rig_div: igrad_q;
VAR h : INTERVAL;
BEGIN
   h:= u/v.f;
   rig_div.f := h;
   rig_div.fq:= -(h*v.fq) / v.f;
END;

{-----------------------------------------------------------------------------}
{ Operators +, -, * and / for one igrad_q and one INTERVAL operand            }
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( VAR u : igrad_q; v : INTERVAL ) igi_add: igrad_q;
BEGIN
   igi_add.f := u.f  + v;
   igi_add.fq:= u.fq
END;

GLOBAL OPERATOR - ( VAR u : igrad_q; v : INTERVAL ) igi_sub: igrad_q;
BEGIN
   igi_sub.f := u.f  - v;
   igi_sub.fq:= u.fq
END;

GLOBAL OPERATOR * ( VAR u : igrad_q; v : INTERVAL ) igi_mul: igrad_q;
BEGIN
   igi_mul.f := u.f * v;
   igi_mul.fq:= u.fq*v;
END;

GLOBAL OPERATOR / ( VAR u : igrad_q; v : INTERVAL ) igi_div: igrad_q;
BEGIN
   igi_div.f := u.f  / v;
   igi_div.fq:= u.fq / v;
END;

{-----------------------------------------------------------------------------}
{ Operators +, -, * and / for one INTERVAL and one igrad_q operand            }
```

```
{-----------------------------------------------------------------------------}

GLOBAL OPERATOR + ( u : INTERVAL; VAR v : igrad_q ) iig_add: igrad_q;
BEGIN
   iig_add.f := u + v.f ;
   iig_add.fq:= v.fq;
END;

GLOBAL OPERATOR - ( u : INTERVAL; VAR v : igrad_q ) iig_sub: igrad_q;
BEGIN
   iig_sub.f := u - v.f;
   iig_sub.fq:= -v.fq
END;

GLOBAL OPERATOR * ( u : INTERVAL; VAR v : igrad_q ) iig_mul: igrad_q;
BEGIN
   iig_mul.f := u * v.f ;
   iig_mul.fq:= u*v.fq;
END;

GLOBAL OPERATOR / ( u : INTERVAL; VAR v : igrad_q ) iig_div: igrad_q;
VAR h : INTERVAL;
BEGIN
   h:= u/v.f;
   iig_div.f := h;
   iig_div.fq:= -(h*v.fq) / v.f;
END;

{-----------------------------------------------------------------------------}
{   elementary functions for igrad_q                                          }
{-----------------------------------------------------------------------------}

GLOBAL FUNCTION sqr( VAR u : igrad_q ) : igrad_q;
BEGIN
   sqr.f := SQR(u.f);
   sqr.fq:= (2.0*u.f) * u.fq;
END;

GLOBAL FUNCTION sqrt( VAR u : igrad_q ) : igrad_q;
VAR h : INTERVAL;
BEGIN
   h:= SQRT(u.f);
   sqrt.f := h;
   sqrt.fq:= u.fq / (2.0*h);
END;

GLOBAL FUNCTION exp( VAR u : igrad_q ) : igrad_q;
VAR h : INTERVAL;
BEGIN
   h:= EXP(u.f);
   exp.f := h;
   exp.fq:= h * u.fq;
END;

GLOBAL FUNCTION EXP2( VAR u : igrad_q ) : igrad_q;
VAR h : INTERVAL;
BEGIN
   h:= EXP2(u.f);
   exp2.f := h;
   exp2.fq:= ( LN(INTVAL(2.0))*h ) * u.fq;
END;

GLOBAL FUNCTION EXP10( VAR u : igrad_q ) : igrad_q;
VAR h : INTERVAL;
```

```
BEGIN
   h:= EXP10(u.f);
   exp10.f := h;
   exp10.fq:= ( LN(INTVAL(10.0))*h ) * u.fq;
END;

GLOBAL FUNCTION ln( VAR u : igrad_q ) : igrad_q;
BEGIN
   ln.f := ln(u.f);
   ln.fq:= u.fq / u.f;
END;

GLOBAL FUNCTION log2( VAR u : igrad_q ) : igrad_q;
BEGIN
   log2.f := LOG2(u.f);
   log2.fq:= u.fq / ( u.f*LN(INTVAL(2.0)) );
END;

GLOBAL FUNCTION log10( VAR u : igrad_q ) : igrad_q;
BEGIN
   log10.f := LOG10(u.f);
   log10.fq:= u.fq / ( u.f*LN(INTVAL(10.0)) );
END;

GLOBAL FUNCTION sin( VAR u : igrad_q ) : igrad_q;
BEGIN
   sin.f := SIN(u.f);
   sin.fq:= COS(u.f) * u.fq;
END;

GLOBAL FUNCTION cos( VAR u : igrad_q ) : igrad_q;
BEGIN
   cos.f :=   COS(u.f);
   cos.fq:= - SIN(u.f) * u.fq;
END;

GLOBAL FUNCTION tan( VAR u : igrad_q ) : igrad_q;
BEGIN
   tan.f := TAN(u.f);
   tan.fq:= u.fq / SQR(COS(u.f));
END;

GLOBAL FUNCTION cot( VAR u : igrad_q ) : igrad_q;
BEGIN
   cot.f := COT(u.f);
   cot.fq:= u.fq / (-SQR(SIN(u.f)));
END;

GLOBAL FUNCTION arcsin( VAR u : igrad_q ) : igrad_q;
BEGIN
   arcsin.f :=  ARCSIN(u.f);
   arcsin.fq:=  u.fq / SQRT( 1.0 - SQR(u.f) );
END;

GLOBAL FUNCTION arccos( VAR u : igrad_q ) : igrad_q;
BEGIN
   arccos.f :=  ARCCOS(u.f);
   arccos.fq:=  u.fq / (-SQRT( 1.0 - SQR(u.f) ) );
END;

GLOBAL FUNCTION arctan( VAR u : igrad_q ) : igrad_q;
BEGIN
   arctan.f :=  ARCTAN(u.f);
   arctan.fq:=  u.fq / ( 1.0 + SQR(u.f) );
```

```
END;

GLOBAL FUNCTION arccot( VAR u : igrad_q ) : igrad_q;
BEGIN
   arccot.f :=  ARCCOT(u.f);
   arccot.fq:=  u.fq / ( -1.0 - SQR(u.f) );
END;

GLOBAL FUNCTION sinh( VAR u : igrad_q ) : igrad_q;
BEGIN
   sinh.f := SINH(u.f);
   sinh.fq:= COSH(u.f) * u.fq;
END;

GLOBAL FUNCTION cosh( VAR u : igrad_q ) : igrad_q;
BEGIN
   cosh.f :=  COSH(u.f);
   cosh.fq:=  SINH(u.f) * u.fq;
END;

GLOBAL FUNCTION tanh( VAR u : igrad_q ) : igrad_q;
BEGIN
   tanh.f := TANH(u.f);
   tanh.fq:= u.fq / SQR(COSH(u.f));
END;

GLOBAL FUNCTION coth( VAR u : igrad_q ) : igrad_q;
BEGIN
   coth.f := COTH(u.f);
   coth.fq:= u.fq / (-SQR(SINH(u.f)));
END;

GLOBAL FUNCTION arsinh( VAR u : igrad_q ) : igrad_q;
BEGIN
   arsinh.f :=  ARSINH(u.f);
   arsinh.fq:=  u.fq / SQRT( SQR(u.f) + 1.0 );
END;

GLOBAL FUNCTION arcosh( VAR u : igrad_q ) : igrad_q;
BEGIN
   arcosh.f :=  ARCOSH(u.f);
   arcosh.fq:=  u.fq / SQRT( SQR(u.f) - 1.0 );
END;

GLOBAL FUNCTION artanh( VAR u : igrad_q ) : igrad_q;
BEGIN
   artanh.f :=  ARTANH(u.f);
   artanh.fq:=  u.fq / ( 1.0 - SQR(u.f) );
END;

GLOBAL FUNCTION arcoth( VAR u : igrad_q ) : igrad_q;
BEGIN
   arcoth.f :=  ARCOTH(u.f);
   arcoth.fq:=  u.fq / ( 1.0 - SQR(u.f) );
END;

END.
```

Concluding this section we mention as in the case of the itaylor and the igradrev modules that we can convert the module by only a few simple changes into a module rgq_ari for the computation of gradients times fixed vectors using the REAL data type, or to a module cgq_ari using the COMPLEX data type or even to a module cigq_ari ! ¡=== SUBINDEX! using the CINTERVAL data type (i.e. complex intervals).

Apart from renaming the module the following changes should be made:

- change the `USE`-clause; no `USE` is necessary for REAL, `USE c_ari` is needed for COMPLEX and `USE i_ari,c_ari,ci_ari` are needed for CINTERVAL.
- change all occurences of the data type INTERVAL to REAL, COMPLEX or CINTERVAL as desired. The name of the data type igrad_q should be changed to rgrad_q, cgrad_q or cigrad_q for clarity.
- discard the calls to the intrinsic function INTVAL in the cases of REAL and COMPLEX.
- operators with mixed operand types (igrad_q, REAL, INTERVAL, COMPLEX and/or CINTERVAL) may be added or deleted as desired.

There are however versions of PASCAL-XSC where many complex standard functions are not implemented (and also not those for complex intervals). With these versions only the basic arithmetic will work in complex, the standard functions will not.

5.3.5 Test Results

As a simple test program for our module igq_ari we compute the function value $f(x)$ and the value of the directional derivative $\nabla f(x) \cdot q$ in two different ways: (i) by use of automatic differentiation as implemented in our module and (ii) directly by computing the gradient by hand and using an explicit expression for $\nabla f(x) \cdot q$.

In the program the following test function $f : I\!R^n \to I\!R$ is used:

$$f(x) = \left(\sum_{i=1}^{n} \sin x_i^2 \right)^{\frac{1}{2}}$$

whose gradient is

$$\nabla f(x) = \frac{1}{f(x)} (x_1 \cos x_1^2, \ldots, x_n \cos x_n^2) \ .$$

Thus the derivative in direction of a vector $q \in I\!R^n$ is

$$\nabla f(x) \cdot q = \frac{1}{f(x)} \sum_{i=1}^{n} q_i x_i \cos x_i^2 \ .$$

```
PROGRAM igq_test;

USE timer;
USE i_ari,mvi_ari,igq_ari;

VAR n : INTEGER;
```

```
FUNCTION f_autodiff( VAR x : igq_vector ) : igrad_q;
VAR i : INTEGER;
    s : igrad_q;
BEGIN
   s:= 0;
   FOR i:= LB(x) TO UB(x) DO  s:= s + sin(x[i]*x[i]);
   f_autodiff:= sqrt( s );
END;

FUNCTION f_direct( VAR x,q : IVECTOR ) : igrad_q;
VAR i   : INTEGER;
    f,s : INTERVAL;
BEGIN
   s:= 0;
   FOR i:= LB(x) TO UB(x) DO  s:= s + SIN(x[i]*x[i]);
   f:= SQRT(s);
   s:= 0;
   for i:= LB(x) TO UB(x) DO  s:= s + x[i]*COS(x[i]*x[i]) * q[i];
   f_direct.f := f;
   f_direct.fq:= s/f;
END;

PROCEDURE main;
VAR x    : igq_vector[1..n];
    fx   : igrad_q;
    xr,q : IVECTOR[1..n];
    i    : INTEGER;
BEGIN
   FOR i:= 1 TO n DO BEGIN
                       xr[i]:= i*(n-1)/(n*n);
                        q[i]:= i - n/2;
                     END;
   init_timer;
   x := igradq( xr, q );
   fx:= f_autodiff(x);
   writeln( 'automatic differentiation:');
   writeln( 'time = ', get_timer, ' msec' );
   writeln( '       f(x) = ', fx.f  );
   writeln( 'grad(f(x))*q = ', fx.fq );
   writeln;

   init_timer;
   fx:= f_direct( xr, q );
   writeln( 'direct computation:');
   writeln( 'time = ', get_timer, ' msec' );
   writeln( '       f(x) = ', fx.f  );
   writeln( 'grad(f(x))*q = ', fx.fq );
   writeln;
END;

BEGIN
   REPEAT
      write('n = '); read(n);
      IF n > 0 THEN main;
   UNTIL n <= 0;
END.
```

A sample output of the program shows that the computation of the directional derivative with the help of our module is just as expensive as the direct computation of $f(x)$ and $\nabla f(x) \cdot q$ using an optimal expression for $\nabla f(x)$. The advantage of automatic differentiation of course is, that we do *only* need the expression for f itself and nothing else. The execution times in this output were measured by use of the standard PASCAL-XSC timer module and are printed in milliseconds. The computation was carried out on an HP 720/9000.

```
n = 10
automatic differentiation:
time = 130 msec
        f(x) = [  1.716840615831126E+000,  1.716840615831128E+000 ]
grad(f(x))*q = [   4.32786519578529E+000,   4.32786519578530E+000 ]

direct computation:
time = 110 msec
        f(x) = [  1.716840615831126E+000,  1.716840615831128E+000 ]
grad(f(x))*q = [   4.32786519578529E+000,   4.32786519578530E+000 ]

n = 100
automatic differentiation:
time = 1040 msec
        f(x) = [   5.55985103995079E+000,   5.55985103995081E+000 ]
grad(f(x))*q = [   1.02699467819120E+002,   1.02699467819121E+002 ]

direct computation:
time = 1030 msec
        f(x) = [   5.55985103995079E+000,   5.55985103995081E+000 ]
grad(f(x))*q = [   1.02699467819120E+002,   1.02699467819121E+002 ]

n = 1000
automatic differentiation:
time = 10480 msec
        f(x) = [    1.7611285963826E+001,    1.7611285963827E+001 ]
grad(f(x))*q = [    3.1468139065330E+003,    3.1468139065333E+003 ]

direct computation:
time = 10270 msec
        f(x) = [    1.7611285963826E+001,    1.7611285963827E+001 ]
grad(f(x))*q = [    3.1468139065330E+003,    3.1468139065333E+003 ]

n = 0
```

Chapter 6

Interval Arithmetic in Staggered Correction Format

Real intervals are usually realized on a computer in such a way that their bounds are chosen as machine numbers, i.e. as floating-point numbers with a fixed precision. Then interval arithmetic is implemented by use of floating-point operations with directed rounding and with results of the same precision. If the machine has floating-point hardware with directed rounding then this kind of interval arithmetic can run at full hardware speed.

Alternatively, sometimes multiple precision real data types are used, again with fixed or with variable precision, especially if ill conditioned problems have to be solved or if there are very high requirements for accuracy of the results. In this case using fast floating-point hardware is no longer possible on traditional hardware since these multiple precision arithmetic operations are usually simulated in integer arithmetic on the computer.

In this chapter we discuss an approach located between these two extremes and which can take full advantage of floating-point hardware provided *only one additional operation* – the *exact scalar product* with a *long accumulator* – is added to the hardware. We represent an interval as a sum of n floating-point numbers $a_i,\ i = 1, \ldots, n$ (which are stored separately) plus one interval A with floating-point bounds (a so called *staggered correction format*). Then all four basic operations $+, -, *, /$ as well as $\sqrt{\ }$ can be efficiently implemented using sums and scalar products of floating-point numbers which are computed exactly in the long accumulator. Also algorithms for standard functions can easily be implemented on the basis of these arithmetical operations.

Of course, using a long accumulator in PASCAL–XSC is no problem, since there the standard data type *DOTPRECISION* is available for this purpose

6.1 Introduction

When implementing an interval arithmetic on a computer it is quite natural (and also the most common implementation), to represent real intervals as *machine intervals*; these are intervals whose bounds are machine numbers, i.e., they are floating-point numbers with a *fixed precision*. Then interval arithmetic can be implemented by use of floating-point operations with directed rounding and with results of the same precision. Thus, interval arithmetic can take full advantage of existing floating-point hardware if this hardware is able to perform directed roundings. This is also the way interval arithmetic is implemented in PASCAL–XSC.

However, in cases where the accuracy of the results is insufficient or no results can be obtained at all due to poorly conditioned problems, it is desirable to have an interval arithmetic of higher precision. Such an arithmetic can easily be obtained if the underlying floating-point system is replaced by a *multiple precision* real arithmetic

as e.g. in [205]. But then the basic operations are generally no longer available in hardware since multiple precision arithmetic has to be simulated by integer arithmetic on the computer. Therefore, execution times will increase drastically if a program uses such a high precision interval arithmetic.

In PASCAL–XSC we have the possibility to solve this problem in a way which can result in execution times closer to those of pure floating-point hardware but still offering a highly increased precision for critical computations. The reason for this is that we have a reliable floating-point arithmetic (i.e. IEEE arithmetic according to the IEEE standard, [143]) and, in addition, we also have the data type *DOTPRECISION* representing a long accumulator. This allows us to implement a new data type *istaggered* for a multiple precision interval arithmetic. With a long accumulator implemented in hardware this *istaggered* arithmetic will run very efficiently and fast using floating-point hardware resources. Such a hardware implementation of a long accumulator has already been developed and will probably soon be available for a large variety of computers ([32], [33]). But even if we simulate the long accumulator in software only, as it is done in PASCAL–XSC at the time being, our approach to higher precision still results in similarly fast codes as in the case of usual high precision arithmetic, which is also simulated in software only.

The high precision data type *istaggered* represents intervals as the sum of n floating-point numbers a_i, $i = 1, \ldots, n$ (which are stored separately) plus one interval A with floating-point bounds (a so called *staggered correction format*, see [306]). Then all four basic operations $+, -, *, /$ and also $\sqrt{\ }$ can be efficiently implemented using sums and scalar products of floating-point numbers which can be computed exactly in the long accumulator. Also all commonly used elementary functions can be implemented without difficulties for this data format. For simplicity, however, here we only discuss arithmetic operations and $\sqrt{\ }$ in more detail and sketch only what has to be done for standard functions.

6.2 Theoretical Background

Let S be the set of all machine numbers on a computer. By $I\mathbb{R}$ we denote the set of all intervals over $\mathbb{R}$ and by IS we denote the set of all *machine intervals*, i.e., all intervals $X = [\underline{x}, \overline{x}]$ with $\underline{x}, \overline{x} \in S$. In this section, we define an interval in *staggered correction format* (or shorter *staggered format*); additionally, we develop algorithms for the four basic operations as well as the square root for this staggered correction data type.

Definition 1 (Staggered Correction Format)
Let $x_i \in S, i = 1, \ldots, n,\ n \geq 0$, be machine numbers and $X \in IS$ a machine interval. Then

$$x = \sum_{i=1}^{n} x_i + X \tag{6.1}$$

is called an interval in staggered correction format *of length n. The set of all intervals in staggered correction format of length n is denoted by IS_n.*

Remark:

1. Sometimes we will refer to x_i as the *i-th component* and to X as the *interval component* of a given staggered interval. Of course, in this definition the representation of an interval in staggered correction format is *not unique* in general. In most cases we even may have staggered representations of different lengths for the same interval $x \in I\mathbb{R}$ as, e.g., any sum of zeros plus an interval X for all $X \in IS$. However, this will not present any difficulties, in fact, all calculations will be carried out by use of the long accumulator which is nothing but a long fixed point register and therefore provides a mean for the unique representation of intermediate results.

2. We explicitly allow n to be zero in Definition 1. Then the sum in (6.1) is an empty sum with value zero. In this case we have $IS_0 = IS$ and the staggered correction representation of an interval in IS_0 is unique. Nevertheless, in our implementation we will require $n \geq 1$ since this will make some parts of the implementation shorter.

3. It is desirable that the absolute values are ordered as $|x_1| > \ldots > |x_n| > |X|$ and that the exponents of two successive summands x_i, x_{i+1}, differ at least by l, where l is the mantissa length. We then say that the mantissas in the staggered correction form do not overlap. In this case the interval x is represented with an optimal precision of about $(n+1)l$ mantissa digits. However, this is not a necessary assumption concerning this format: we could even allow all components x_i as well as X to be of the same order of magnitude such that no precision is gained by storing x as a staggered interval. We still could perform all operations and computations but, of course, we would then face a considerable loss of efficiency. Therefore, the non-overlapping property is very desirable, and the operations on IS_n should be written such that they produce results with this property. Our implementation will achieve this property.

4. One advantage of the staggered format is the fact that we can store a number x with high precision using only a few components. If e.g. x is of the form $x = 1 + \epsilon$ with a machine number ϵ then x can be stored *exactly* even if ϵ is very small, $|\epsilon| < 10^{-m}$, m large, say. In this case we can store the number x with a precision of m digits by using only a staggered length of $n = 1$ i.e., one real and one interval are sufficient to enclose x.

The exact results of the arithmetic operations on staggered intervals are defined as usual in interval analysis: if x and y are two staggered intervals, we treat them as elements of $I\!R$ and define the operations as they are defined there:

Definition 2 (Exact operations on staggered intervals)
Let x and y be intervals in a staggered correction format.
Then $x \circ y := \{\zeta \mid \zeta = \xi \circ \eta, \xi \in x, \eta \in y\}$, in the case of $\circ \in \{+, -, *, /\}$, if $0 \notin y$ for $\circ = /$ and $\sqrt{x} := \{\zeta \mid \zeta = \sqrt{\xi}, \xi \in x\}$, if $\inf(x) \geq 0$.

Of course, we cannot expect the arithmetic operations to produce *exact* results on a computer since we still are limited by the machine precision, i.e. , the length of the long accumulator in this case. However, for any operation $z = x \circ y$ or $z = \sqrt{x}$, we must require that the staggered interval z is a superset of the true mathematical result of $x \circ y$, as is also the case for ordinary machine interval arithmetic: $x \circ y \subseteq z$ and $\sqrt{x} \subseteq z$. Here we do not require the result z to be optimal in the sense that it is the smallest staggered interval of some prescribed length n containing the true mathematical value of $x \circ y$ or $\sqrt{x}$; rather, we are satisfied with a compromise between the tightness of the enclosure z and the ease and efficiency of the implementation.

We desire the operations to accept staggered operands of any (perhaps differing) length and to produce their result in a staggered format whose length can be prescribed independently of that of the operands. This can be achieved conveniently if all operations follow the same pattern as far as possible: first we compute upper and lower bounds for the true result and store them in two long accumulators. Then by use of proper rounding, these accumulators are converted to a staggered interval of the desired length. Division and square root will be the only operations differing from this pattern.

First we comment on the two unary operations $+$ and $-$. If x is a staggered interval of length n and if n is also the prescribed result length of $+x$ and $-x$, then $+$ is the identity where no operation is necessary, and $-$ is the operation reversing the signs of all components of x. However, if the prescribed result length is $m \neq n$ then there are two cases. The first case, where $m > n$, is similar to the case of $m = n$: all components of x must be copied (with or without a reversal the signs) and only the additional $m-n$ real components of the result have to be set to zero. The second case, however, where $m < n$ requires the execution of some nontrivial operations; we define two accumulators lo and up to store the bounds of x: $lo := \sum_{i=1}^{n} x_i + \underline{X}$ and $up := \sum_{i=1}^{n} x_i + \overline{X}$. Then these accumulators are converted to a staggered interval of length m by the same procedure which is used to obtain the results for the other operations.

This *conversion* of two accumulators lo and up to an staggered interval x of length n is now outlined. Let lo and up be two long accumulators and $n \geq 0$ a prescribed staggered length. Then lo and up are converted to x by use of the following algorithm Convert. Here, lo and up are long accumulators, xl, xu, x_i are reals, X is an interval, i, n are integers, and *stop* is a boolean variable. We list algorithm Convert already in a more algorithmic form since it is less mathematical in nature.

Conversion of two accumulators lo,up to an istaggered :
Input: accumulators lo and up and result length n
Output: staggered interval x

$$
\begin{array}{l}
stop := false \\
i := 0 \\
repeat \\
\quad i := i + 1 \\
\quad xl := \square lo \\
\quad xu := \square up \\
\quad if\ xl = xu\ then\ x_i := xl \\
\qquad\qquad\qquad\quad lo := lo - xl \\
\qquad\qquad\qquad\quad up := up - xu \\
\qquad\qquad\quad else\ x_i := 0 \\
\qquad\qquad\qquad\quad stop := true \\
until\ stop\ or\ i = n \\
for\ i := i + 1\ to\ n\ do\ x_i := 0 \\
X := [\bigtriangledown lo, \bigtriangleup up]
\end{array}
\tag{6.2}
$$

Here, $\square, \bigtriangledown, \bigtriangleup$ represent the roundings of the accumulator to the nearest, next lower and next upper machine number, respectively. This algorithm reads successively real numbers *xl* and *xu* from the accumulators and, as long as they are equal and the staggered length has not yet been reached, they are assigned to x_i and subtracted from the accumulators. When *xl* and *xu* are different or when the staggered length n for the result x is reached, then the remaining values in the accumulators are converted to the interval component X of x using proper rounding. In the case that some x_i's are not yet defined, they are set to zero. This algorithm has the property that the real components of x do not overlap in the sense described above. Thus, this property will also hold true for the results of the following operators where algorithm Convert is used.

Next, we consider *addition* and *subtraction* of staggered intervals. There we simply add (subtract) the lower and upper bounds of two staggered operands x and y into two accumulators and convert them to the result z using the above algorithm Convert. Here, *lo* and *up* are long accumulators, x_i, y_i are reals, X, Y are intervals, and z is the resulting staggered interval.

In the case of addition we have

$$
\begin{array}{lcl}
up & := & \sum_{i=1}^{n_x} x_i + \sum_{i=1}^{n_y} y_i \\
lo & := & up + \underline{X} + \underline{Y} \\
up & := & up + \overline{X} + \overline{Y} \\
z & := & \mathsf{Convert}(\ lo,\ up\ )
\end{array}
\tag{6.3}
$$

and analogously in the case of subtraction:

$$
\begin{array}{lcl}
up & := & \sum\limits_{i=1}^{n_x} x_i \; - \; \sum\limits_{i=1}^{n_y} y_i \\
lo & := & up + \underline{X} - \overline{Y} \\
up & := & up + \overline{X} - \underline{Y} \\
z & := & \mathsf{Convert}\,(\ lo,\ up\ )
\end{array}
\tag{6.4}
$$

Obviously, addition and subtraction are straightforward. *Multiplication* however can be implemented in various ways yielding different results because of the subdistributive law of interval arithmetic. Let $x = \sum_{i=1}^{n_x} x_i + X$ and $y = \sum_{j=1}^{n_y} y_j + Y$ be two staggered intervals then, due to subdistributivity, we have:

$$
\begin{array}{lcl}
xy & = & (\sum\limits_{i=1}^{n_x} x_i + X) \cdot (\sum\limits_{j=1}^{n_y} y_j + Y) \\
 & \subseteq & \sum\limits_{i=1}^{n_x} \sum\limits_{j=1}^{n_y} x_i y_j + X \cdot \sum\limits_{j=1}^{n_y} y_j + Y \cdot \sum\limits_{i=1}^{n_x} x_i + XY \\
 & \subseteq & \sum\limits_{i=1}^{n_x} \sum\limits_{j=1}^{n_y} x_i y_j + \sum\limits_{j=1}^{n_y} X y_j + \sum\limits_{i=1}^{n_x} Y x_i + XY.
\end{array}
\tag{6.5}
$$

This seems to suggest that better results can be obtained from using the second line in (6.5) than from using the third line. However, in order to compute the products $X \sum_{j=1}^{n_y} y_j$ and $Y \sum_{i=1}^{n_x} x_i$ we first have to round the sums to machine intervals, $\Diamond(\sum_{j=1}^{n_y} y_j)$ and $\Diamond(\sum_{i=1}^{n_x} x_i)$. As a consequence of these additional roundings, however, now the second line in (6.5) often yields *coarser* enclosures than the third line.

Therefore, we choose line three in (6.5) as the basis of our multiplication algorithm. Here, again, lo and up are long accumulators, x_i, y_i are reals, X, Y are intervals, and z is the resulting staggered interval.

$$
\begin{array}{lcl}
lo & := & \sum\limits_{i=1}^{n_x} \sum\limits_{j=1}^{n_y} x_i y_j \\
up & := & lo \\
[lo, up] & := & [lo, up] + \sum\limits_{j=1}^{n_y} X y_j + \sum\limits_{i=1}^{n_x} Y x_i + XY \\
z & := & \mathsf{Convert}\,(\ lo,\ up\ )
\end{array}
\tag{6.6}
$$

Up to now, we were able to compute bounds for the results of our operations in two accumulators and then round these to an interval staggered format. This can no longer be carried out conveniently in the case of the *division* of two staggered correction intervals x and y. Rather, we will apply an iterative algorithm computing successively the n_z real components z_i of the quotient x/y.

In order to compute this approximation $\sum_{i=1}^{n_z} z_i$, we start with $z_1 = \Box m(x)/\Box m(y)$; here $m(a)$ represents a point selected in a, e.g. the midpoint and $\Box$ is the rounding to S. Now, we proceed inductively: if we have an approximation $\sum_{i=1}^{k} z_i$, then we can compute a next summand z_{k+1} by use of

$$
z_{k+1} = \Box \left(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{k} y_i z_j \right) / \Box(m(y)),
\tag{6.7}
$$

where the numerator is computed exactly using a long accumulator and is rounded only once to S. The division is performed in ordinary floating-point arithmetic.

As in the previous operations, this iteration guarantees that the z_i do not overlap since the defect (i.e. the numerator in (6.7)) of each approximation $\sum_{i=1}^{k} z_i$ is computed with only one rounding.

Now, the interval component Z of the result z may be computed as follows:

$$Z = \Diamond \Big(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{n_z} y_i z_j + X - \sum_{j=1}^{n_z} z_j Y \Big) / \Diamond(y), \tag{6.8}$$

where $\Diamond$ is the rounding to an interval in IS.

It is not difficult to see that $z = \sum_{i=1}^{n_z} z_i + Z$ as computed from (6.7) and (6.8) is a superset of the exact range $\{\xi/\eta \mid \xi \in x, \eta \in y\}$; in fact, for all $\alpha \in X, \beta \in Y$ we have the identity:

$$\frac{\sum\limits_{i=1}^{n_x} x_i + \alpha}{\sum\limits_{i=1}^{n_y} y_i + \beta} = \sum_{j=1}^{n_z} z_i + \frac{\sum\limits_{i=1}^{n_x} x_i + \alpha - \sum\limits_{i=1}^{n_y} \sum\limits_{j=1}^{n_z} y_i z_j - \sum\limits_{j=1}^{n_z} z_j \beta}{\sum\limits_{i=1}^{n_y} y_i + \beta}.$$

An interval evaluation of this expression for $\alpha \in X$ and $\beta \in Y$ shows immediately that the exact range of x/y is contained in $\sum_{j=1}^{n_z} z_j + Z$, which is computed using (6.7) and (6.8).

Now, it is clear how to get the result z for the division of two *istaggered* variables x/y by the following three computation steps:

1. $z_1 := m(x)/m(y)$
2. compute real parts z_{k+1} from (6.7) for $k = 0, \ldots, n_z - 1$ (6.9)
3. compute interval part Z from (6.8)

An algorithm for the *square root* can be obtained analogously as in the case of the division. We compute iteratively as follows the $z_i, i = 1, \ldots, n_z$ of the approximation part:

$$\begin{aligned} z_1 &= \sqrt{\Box(x)} \\ z_{k+1} &= \Box \Big(\sum_{i=1}^{n_x} x_i - \sum_{i,j=1}^{k} z_i z_j \Big) / (2 z_1). \end{aligned} \tag{6.10}$$

This guarantees again that the z_i do not overlap since in the numerator of (6.10) the defect of the approximation $\sum_{i=1}^{k} z_i$ is computed with one rounding only. Now, the interval part Z is computed by use of

$$Z = \frac{\Diamond \Big(\sum\limits_{i=1}^{n_x} x_i - \sum\limits_{i,j=1}^{n_z} z_i z_j + X \Big)}{\sqrt{\Diamond(x)} + \Diamond \Big(\sum\limits_{i=1}^{n_z} z_i \Big)}. \tag{6.11}$$

As in the case of the division, it is easy to see that $\sum_{i=1}^{n_z} z_i + Z$ as computed from (6.10) and (6.11) is a superset of the exact range $\{\sqrt{\xi} \mid \xi \in x\}$; in fact, for all $\gamma \in X$ we have the identity:

$$\sqrt{\sum_{i=1}^{n_x} x_i + \gamma} = \sum_{j=1}^{n_z} z_j + \frac{\sum\limits_{i=1}^{n_x} x_i + \gamma - \sum\limits_{i,j=1}^{n_z} z_i z_j}{\sqrt{\sum\limits_{i=1}^{n_x} x_i + \gamma} + \sum\limits_{j=1}^{n_z} z_j}.$$

Now, we see that the square root z of a staggered interval x can be computed by the following three steps very similar to the case if division:

$$\begin{array}{ll} 1. & z_1 := \sqrt{\Box x} \\ 2. & \text{compute real parts } z_{k+1} \text{ from (6.10) for } k = 0, \ldots, n_z - 1 \\ 3. & \text{compute interval part Z from (6.11)} \end{array} \tag{6.12}$$

Remark: In the cases of addition, subtraction, and multiplication we have non-overlapping real components z_i since the algorithm Convert is used here. In the cases of division and the square root this property holds true since the defects of the approximations are computed with one rounding only. Nevertheless, it may still be possible that we obtain overlapping in the sense that some z_i will be zero; this may happen e.g. if we choose n_z too large such that underflow occurs when a long accumulator is converted to a staggered interval. It may also happen that we obtain a non-overlapping approximation part $\sum_{i=1}^{n_z} z_i$, which, however, overlaps with the interval part Z in the sense that the absolute value of Z exceeds those of some of the z_i's. This is due to the fact that we allow an arbitrary result length n_z to be specified prior to the computation of the operation such that n_z may be larger than the best possible accuracy of the result (which is limited, of course, by the accuracy of the operands).

6.3 Algorithms

For the following algorithms an interval in staggered correction form is assumed to be represented by the following PASCAL–XSC data type:

```
TYPE  istaggered  = DYNAMIC ARRAY [*] OF REAL;
```

The index range of an *istaggered* variable x must always go from -1 to n, n ≥ 1, where the elements with index 1,...,n, are the real components x_i and the elements with index -1 resp. 0 are the lower resp. upper bounds of the interval part X of x.

Also, in the following algorithms we assume that the actual length n of the *istaggered* variables is contained in a global variable *stagg_prec*. The value of this variable will be used to determine the length of the staggered variables which are the results of the lgorithms.

The staggered length of the operands of the algorithms need not be separate parameters since the upper bounds of the arrays representing the *istaggered* variables can be obtained in PASCAL–XSC by the standard function *ub*( ). Thus for an *istaggered* variable x the staggered length $n_x = ub(\text{x})$.

Algorithm 6.1: Convert (lo, up) {function}

{Convert two *DOTPRECISIONS* lo, up to an *istaggered*}
input: *DOTPRECISIONS* lo and up and result length *stagg_prec*
output: *istaggered* containing $[lo, up]$ (function's result)

```
stop := false
i := 0
repeat
   i := i + 1
   xl := □lo
   xu := □up
   if xl = xu  then  Convert[i] := xl
                     lo := lo − xl
                     up := up − xu
               else  Convert[i] := 0
                     stop := true
until stop or i = stagg_prec
for i := i + 1 to stagg_prec do Convert[i] := 0
Convert[−1] := ▽lo
Convert[0] := △up
```

Next, we present the algorithms for addition and subtraction according to (6.3) and (6.4). To indicate the use of machine and interval operations we use #-, □- and ◇-signs.

Algorithm 6.2: $+(x, y)$ {operator}

{Add two *istaggereds* x and y}
input: *istaggereds* x and y of length n_x and n_y, respectively
output: *istaggered* of length *stagg_prec* containing $x + y$ (operator's result)

$up := \sum_{i=1}^{n_x} x_i + \sum_{i=1}^{n_y} y_i$
$lo := up + \underline{X} + \underline{Y}$
$up := up + \overline{X} + \overline{Y}$
return Convert (lo, up)

Algorithm 6.3: $-(x, y)$ {operator}

{Subtract two *istaggereds* x and y}
input: *istaggereds* x and y of length n_x and n_y, respectively
output: *istaggered* of length *stagg_prec* containing $x - y$ (operator's result)

$up := \sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} y_i$
$lo := up + \underline{X} - \overline{Y}$
$up := up + \overline{X} - \underline{Y}$
return Convert (lo, up)

The multiplication of two variables in interval staggered format using a computer is performed in a special way for accuracy reasons as described in (6.6).

Algorithm 6.4: $*(x, y)$ {operator}

{Multiply two *istaggereds* x and y}
input: *istaggereds* x and y of length n_x and n_y, respectively
output: *istaggered* of length *stagg_prec* containing $x * y$ (operator's result)

$lo := \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} x_i y_j$
$up := lo$
$[lo, up] := [lo, up] + \sum_{j=1}^{n_y} X y_j + \sum_{i=1}^{n_x} Y x_i + X \cdot Y$
return Convert (lo, up)

The division of two variables in interval staggered format is carried out by an iteration as introduced in (6.9).

Algorithm 6.5: $/(x, y)$ {operator}

{Divide two *istaggereds* x and y}
input: *istaggereds* x and y of length n_x and n_y, respectively
output: *istaggered* z of length *stagg_prec* containing x/y (operator's result)

$n_z := stagg\_prec$
$lo := \sum_{i=1}^{n_x} x_i$
$y_m := \Box m(y)$
$z_1 := (\Box lo)/y_m$
for $k := 2$ **to** n_z **do** $lo := lo - \sum_{j=1}^{n_y} y_j z_{k-1}$
$\quad z_k := (\Box lo)/y_m$
$lo := lo - \sum_{j=1}^{n_y} y_j z_{n_z}$
$up := lo$
$Z := \Diamond([lo, up] + X - \sum_{j=1}^{n_z} z_j Y)/\Diamond y$
return Z

Similar to the division of two variables in interval staggered format the square root is computed in an iterative way (see (6.12)).

Algorithm 6.6: Sqrt (x) {function}

{Square root of an *istaggered* x }
input: *istaggered* x of length n_x
output: *istaggered* z of length *stagg_prec* containing $\sqrt{x}$ (function's result)

$n_z := stagg\_prec$
$lo := \sum_{i=1}^{n_x} x_i$
$z_1 := \sqrt{\Box(lo)}$

$$\textbf{for } k := 2 \textbf{ to } n_z \textbf{ do } \quad lo := lo - 2\sum_{j=1}^{k-2} z_j z_{k-1} - z_{k-1} z_{k-1}$$
$$z_k := (\Box lo)/(2z_1)$$
$$lo := lo - 2\sum_{j=1}^{n_z-1} z_j z_{n_z} - z_{n_z} z_{n_z}$$
$$up := lo$$
$$Z := \Diamond([lo, up] + X)/(\sqrt{\Diamond x} + \Diamond \sum_{i=1}^{n_z} z_i)$$
$$\mathsf{Sqrt} := z$$

Applicability of the Algorithms

Since the arithmetic for staggered correction intervals is based on the long accumulator the maximally available precision is also limited by the precision of this long accumulator. If the underlying floating-point format is IEEE double (as is the case in PASCAL–XSC) we have an exponent range of roughly $10^{\pm 308}$ [143]. Thus the available accuracy depends on the order of magnitude of the numbers which are used in the computation. If the numbers' magnitude is $\approx 10^m$ then we can represent it with a precision of at most $308 + m$ decimal digits (using an appropriate length of the staggered representation).

The algorithm for the square root fails in the case when the first approximation z_1 of the root is zero. However, this flaw can very easily be avoided by introducing some additional checks on x for zeros before starting the algorithm. The implementation in the next section contains e.g. a simple check for x being exactly zero, before Algorithm 6.6 is started.

In order to make the application of our algorithms easy and efficient, some additional operations from interval arithmetic should be supplied such as the computation of an infimum, supremum, diameter or midpoint. Furthermore, conversion routines from and to other data formats as well as operations with mixed operand types and also some comparison operators are useful for the practical work in a programming language. In the module *istagger* in the next section a collection of such additional routines is embedded such that the module can be used with more benefit.

What is still lacking in the module *istagger* are efficient input/output procedures for staggered intervals. These routines are more difficult to implement. Therefore at the moment the use of staggered variables is confined to the internal computations of a program. The final results, however, do benefit from the higher precision, of course. We also should point out that it does not make much sense to print a staggered variable just component-wise (i.e. for $i := 1$ to n do $write(x[i])$). The resulting numbers on the output will be very misleading, since each of the x[i]'s will be rounded individually and independent of the others from the internal binary representation to a decimal representation. Thus each printed component contains a conversion error which makes it impossible to reconstruct the exact value of x.

6.4 PASCAL–XSC Program Code

The following PASCAL–XSC module *istagger* contains an implementation of a staggered interval arithmetic using the algorithms described above. The global type *istaggered* representing a staggered correction interval is chosen to be a dynamic array of reals which contains the real values x_i ($i = 1, \ldots, stagg\_prec$) in the components $1, \ldots, stagg\_prec$ and the lower and upper bound of the interval X in the components -1 and 0 resp.

The global variable *stagg_prec*, which is the prescribed result length n_z for all operations, can be altered as desired during the execution of the program. For this purpose the functions *set_precision* and *get_precision* should be used. The functions and operators will then produce results of the new staggered length *stagg_prec* while accepting any length as input. *stagg_prec* must always be positive (zero is not allowed in this module in order to keep the code somewhat simpler). *stagg_prec* is initialized with the value one in the module's main body.

The local function *convert* is the implementation of Algorithm Convert (Algorithm 6.1). The local auxiliary function *add_i_times_ist* is used to add the product of an interval times a point-*istaggered* to two dotprecision variables.

In addition to the algorithms described in the previous sections we have added some functions, procedures and overloaded assignments which make the module more comfortable to use. We do not describe them all here, since each of these routines is very short and contains a comment stating its purpose.

```
MODULE istagger;

USE i_ari,x_real;

{---------------------------------------------------------------------------}
{  The module istaggered contains a collection of functions and             }
{  operators implementing a high precision interval arithmetic by use of    }
{  the data type istaggered.                                                }
{                                                                           }
{  A variable x of type istaggered represents an interval of the form:      }
{                                                                           }
{          ub(x)                                                            }
{           sum  x[i]  +  [ x[-1], x[0] ]                                   }
{           i=1                                                             }
{---------------------------------------------------------------------------}

GLOBAL TYPE    istaggered  = GLOBAL DYNAMIC ARRAY [*] OF REAL;

GLOBAL VAR     stagg_prec : INTEGER;

{---------------------- precision control functions : ----------------------}

GLOBAL PROCEDURE set_precision( prec : INTEGER );
{ set_precision should be used to set the precision variable stagg_prec }
BEGIN
   IF prec<=0 THEN stagg_prec:= 1
              ELSE stagg_prec:= prec;
END;

GLOBAL FUNCTION get_precision : INTEGER;
{ get_precision returns the current precision stagg_prec }
BEGIN
```

```
   get_precision:= stagg_prec;
END;

GLOBAL FUNCTION change_length ( VAR x : istaggered) :
                                                   istaggered[-1..stagg_prec];
{ converts istaggered x to same format but with new precision stagg_prec }
VAR i,min : INTEGER;
    d     : DOTPRECISION;
BEGIN
   IF stagg_prec>UB(x) THEN min:= UB(x)
                       ELSE min:= stagg_prec;
   FOR i:=   1   TO     min    DO change_length[i]:= x[i];
   FOR i:= min+1 TO stagg_prec DO change_length[i]:= 0.0;
   d:= #( FOR i:= min+1 TO UB(x) SUM ( x[i] ) );
   change_length[-1]:= #<( d + x[-1] );
   change_length[ 0]:= #>( d + x[ 0] );
END;

{----------------------- auxiliary functions : -----------------------------}

GLOBAL FUNCTION inf( VAR x : istaggered ) : istaggered[-1..stagg_prec];
{ infimum of an istaggered x }
BEGIN
   inf:= x;
   inf[0]:= x[-1];
END;

GLOBAL FUNCTION sup( VAR x : istaggered ) : istaggered[-1..stagg_prec];
{ supremum of an istaggered x }
BEGIN
   sup:= x;
   sup[-1]:= x[0];
END;

GLOBAL FUNCTION diam ( VAR x : istaggered ) : REAL;
{ upper bound for the diameter of an istaggered variable }
BEGIN
   diam := x[0] -> x[-1];
END;

GLOBAL FUNCTION mid ( VAR x : istaggered ) : istaggered[-1..stagg_prec];
{ midpoint of an istaggered x in istaggered format }
VAR m : REAL;
BEGIN
   m  := mid( INTVAL(x[-1],x[0]) );
   mid:= x;
   mid[-1]:= m;
   mid[ 0]:= m;
END;

{----------------------- type conversion functions : ------------------------}

GLOBAL FUNCTION ival( VAR x : istaggered ) : INTERVAL;
{ convert an istaggered x to a floating-point interval }
VAR i : INTEGER;
    d : DOTPRECISION;
BEGIN
   d:= #( FOR i:= 1 TO UB(x) SUM( x[i] )  );
   ival:= INTVAL( #<(d + x[-1]), #>( d + x[0] ) );
END;

GLOBAL FUNCTION sintval ( x : REAL ) : istaggered[-1..stagg_prec];
{ convert a REAL x to an istaggered }
VAR i : INTEGER;
```

```
BEGIN
   FOR i:= -1 TO stagg_prec DO sintval[i]:= 0.0;
   sintval[1]:= x;
END;

GLOBAL OPERATOR := ( VAR x : istaggered; y : REAL );
{ assignment of a REAL number y to an istaggered variable x }
VAR i : INTEGER;
BEGIN
   FOR i:= -1 TO stagg_prec DO x[i]:= 0.0;
   x[1]:= y;
END;

GLOBAL FUNCTION sintval ( VAR x : INTERVAL ) : istaggered[-1..stagg_prec];
{ convert INTERVAL x to an istaggered }
VAR i : INTEGER;
BEGIN
   sintval[-1]:= INF(x);
   sintval[ 0]:= SUP(x);
   FOR i:= 1 TO stagg_prec DO sintval[i]:= 0.0;
END;

GLOBAL OPERATOR := ( VAR x : istaggered; VAR y : INTERVAL );
{ assignment of an INTERVAL y to an istaggered x }
VAR i : INTEGER;
BEGIN
   x[-1]:= INF(y);
   x[ 0]:= SUP(y);
   FOR i:= 1 TO stagg_prec DO x[i]:= 0.0;
END;

GLOBAL FUNCTION sintval ( x,y : REAL ) : istaggered[-1..stagg_prec];
{ compose an istaggered from two REALs x,y. x and y may be unordered }
VAR i : INTEGER;
    h : INTERVAL;
BEGIN
   h:= x +* y;
   sintval[-1]:= INF(h);
   sintval[ 0]:= SUP(h);
   FOR i:= 1 TO stagg_prec DO sintval[i]:= 0.0;
END;

{------------------------ some comparisons : --------------------------------}

GLOBAL OPERATOR IN ( x : REAL; VAR y : istaggered ) r_in_ist : boolean;
{ checks if REAL x is contained in istaggered y }
VAR d : DOTPRECISION;
    i : INTEGER;
BEGIN
   d:= #( FOR i:= 1 TO UB(y) SUM ( y[i] ) - x );
   r_in_ist:= ( SIGN( #(d+y[-1]) ) <= 0 ) AND ( SIGN( #(d+y[0]) ) >= 0 );
END;

GLOBAL OPERATOR = ( VAR x,y : istaggered ) eq_ist_ist : boolean;
{ checks if two istaggereds x,y are equal }
VAR d : DOTPRECISION;
    i : INTEGER;
BEGIN
   d:= #(   FOR i:= 1 TO UB(x) SUM ( x[i] )
          - FOR i:= 1 TO UB(y) SUM ( y[i] )  );
   eq_ist_ist:= ( SIGN( #(d+x[-1]-y[-1]) ) = 0 ) AND
                ( SIGN( #(d+x[0] -y[0])  ) = 0 );
END;
```

```
GLOBAL OPERATOR <> ( VAR x,y : istaggered ) neq_ist_ist : boolean;
{ checks if two istaggereds x,y are not equal }
BEGIN
   neq_ist_ist:= NOT ( x = y );
END;

{---------------------- istaggered arithmetic : ----------------------------}

{---------------------------------------------------------------------------}
{ two local auxiliary functions for staggered arithmetic :                  }
{---------------------------------------------------------------------------}

FUNCTION convert ( VAR lo,up : DOTPRECISION ) : istaggered[-1..stagg_prec];
{ convert the contents of the two DOTPRECISION lo,up to an istaggered }
{ (auxiliary function for most of the following arithmetic operators) }
VAR i      : INTEGER;
    xl,xu : REAL;
    stop   : BOOLEAN;
BEGIN
   stop:= FALSE;
   i:= 0;
   REPEAT
      i := i + 1;
      xl:= #*( lo );
      xu:= #*( up );
      IF xl=xu THEN BEGIN
                       convert[i]:= xl;
                       lo:= #( lo - xl );
                       up:= #( up - xu );
                    END
               ELSE BEGIN
                       convert[i]:= 0.0;
                       stop:= TRUE;
                    END;
   UNTIL stop OR (i=stagg_prec);
   FOR i:= i+1 TO stagg_prec DO convert[i]:= 0.0;
   convert[-1]:= #<( lo );
   convert[ 0]:= #>( up );
END;

PROCEDURE add_i_times_ist( VAR x : INTERVAL; VAR y : istaggered;
                           VAR lo,up : DOTPRECISION );
{ add the product x*y of an interval x and an istaggered y to the contents }
{ of the DOTPRECISIONs lo and up.                                          }
{ The interval part of y must be zero ! (i.e. y[-1] = y[0] = 0.0)          }
VAR i : INTEGER;
BEGIN
   FOR i:= 1 TO UB(y) DO
      IF y[i]>0.0 THEN BEGIN
                          lo:= #( lo + INF(x)*y[i] ) ;
                          up:= #( up + SUP(x)*y[i] ) ;
                       END
                  ELSE BEGIN
                          lo:= #( lo + SUP(x)*y[i] ) ;
                          up:= #( up + INF(x)*y[i] ) ;
                       END;
END;


{-------------------------------------------------------------------------}
{ unary operators +, - for istaggered :                                   }
{-------------------------------------------------------------------------}

GLOBAL OPERATOR  + ( VAR x : istaggered ) ist_uADD : istaggered[-1..UB(x)];
```

```
{ unary plus of an istaggered : does  n o t  change precision of x ! }
BEGIN
   ist_uADD:= x;
END;

GLOBAL OPERATOR  - ( VAR x : istaggered ) ist_uSUB : istaggered[-1..UB(x)];
{ unary minus of an istaggered : does  n o t  change precision of x ! }
VAR i : INTEGER;
BEGIN
   FOR i:= 1 TO UB(x) DO ist_uSUB[i]:= -x[i];
   ist_uSUB[-1]:= -x[ 0];
   ist_uSUB[ 0]:= -x[-1];
END;

{---------------------------------------------------------------------------}
{ operators +, -, *, / for two istaggereds :                                }
{---------------------------------------------------------------------------}

GLOBAL OPERATOR  + ( VAR x,y : istaggered )
                                     ist_ist_ADD : istaggered[-1..stagg_prec];
{ addition x + y of two istaggereds x,y }
VAR lo,up  : DOTPRECISION;
    i      : INTEGER;
BEGIN
   up:= #(   FOR i:= 0 TO UB(x) SUM ( x[i] )     { to avoid multiple #-calls }
           + FOR i:= 0 TO UB(y) SUM ( y[i] ) );  { up is computed directly   }
   lo:= #( up - x[0] - y[0] + x[-1] + y[-1] );   {--------------------------}
   ist_ist_ADD:= convert( lo,up );
END;

GLOBAL OPERATOR  - ( VAR x,y : istaggered )
                                     ist_ist_SUB : istaggered[-1..stagg_prec];
{ subtraction x - y of two istaggereds x,y }
VAR lo,up  : DOTPRECISION;
    i      : INTEGER;
BEGIN
   up:= #( FOR i:= 0 TO UB(x) SUM ( x[i] ) -           { to avoid multiple }
           FOR i:= 1 TO UB(y) SUM ( y[i] ) - y[-1]  );  { #-calls up is     }
   lo:= #( up - x[0] + y[-1] + x[-1] - y[0] );         { computed directly }
   ist_ist_SUB:= convert( lo,up );                     {-------------------}
END;

GLOBAL OPERATOR  * ( VAR x,y : istaggered )
                                     ist_ist_MUL : istaggered[-1..stagg_prec];
{ multiplication x * y of two istaggereds x,y }
VAR lo,up : DOTPRECISION;
    i,j   : INTEGER;
    C     : INTERVAL;
BEGIN
   lo := #(   FOR i:= 1 TO UB(x) SUM
            ( FOR j:= 1 TO UB(y) SUM ( x[i]*y[j] ) )  );
   up := lo;
   add_i_times_ist( INTVAL(x[-1],x[0]), y, lo,up );          { sum( X * y) }
   add_i_times_ist( INTVAL(y[-1],y[0]), x, lo,up );          { sum( Y * x) }
   C:= INTVAL(x[-1],x[0])*INTVAL(y[-1],y[0]);                { C = X * Y   }
   lo:= #( lo + C.INF ) ;                                    {-------------}
   up:= #( up + C.SUP ) ;
   ist_ist_MUL:= convert( lo,up );
END;

GLOBAL OPERATOR / ( VAR x,y : istaggered )
                                     ist_ist_DIV : istaggered[-1..stagg_prec];
{ division x / y of two istaggereds x,y }
VAR lo,up : DOTPRECISION;
```

```
    i,j   : INTEGER;
    z     : istaggered[-1..stagg_prec];
    ym    : REAL;
    C     : INTERVAL;
BEGIN
   lo  := # ( FOR i:= 1 TO UB(x) SUM ( x[i] ) );
   ym  := #*( FOR i:= 1 TO UB(y)                      { ym is midpoint of y }
              SUM ( y[i] ) + 0.5*y[0] + 0.5*y[-1] );
   z[1]:= #*(lo) / ym;
   FOR i:= 2 TO stagg_prec DO
   BEGIN
      lo  := #( lo - FOR j:= 1 TO UB(y) SUM ( y[j]*z[i-1] ) );
      z[i]:= #*(lo) / ym;
   END;
   lo:= #( lo - FOR j:= 1 TO UB(y) SUM ( y[j]*z[stagg_prec] ) );
   up:= lo;
   add_i_times_ist( INTVAL(-y[0],-y[-1]), z, lo,up );        { sum( Y * z ) }
   C     := INTVAL( #<(lo+x[-1]),#>(up+x[0]) ) / ival(y);
   z[-1] := C.INF;
   z[ 0] := C.SUP;
   ist_ist_DIV:= z;
END;

{---------------------------------------------------------------------------}
{ operators +, -, *, / for REAL and istaggered :                            }
{---------------------------------------------------------------------------}

GLOBAL OPERATOR  + ( x : REAL; VAR y : istaggered )
                                      r_ist_ADD : istaggered[-1..stagg_prec];
{ addition x + y of a REAL x and an istaggered y }
VAR lo,up : DOTPRECISION;
    i     : INTEGER;
BEGIN
   up:= #( x + FOR i:= 0 TO UB(y) SUM ( y[i] ) );
   lo:= #( up - y[0] + y[-1] );
   r_ist_ADD:= convert( lo,up );
END;

GLOBAL OPERATOR  - ( x : REAL; VAR y : istaggered )
                                      r_ist_SUB : istaggered[-1..stagg_prec];
{ subtraction x - y of a REAL x and an istaggered y }
VAR lo,up : DOTPRECISION;
    i     : INTEGER;
BEGIN
   up:= #( x - FOR i:= 1 TO UB(y) SUM ( y[i] ) - y[-1] );
   lo:= #( up - y[0] + y[-1] );
   r_ist_SUB:= convert( lo,up );
END;

GLOBAL OPERATOR  * ( x : REAL; VAR y : istaggered )
                                      r_ist_MUL : istaggered[-1..stagg_prec];
{ multiplication x * y of a REAL x and an istaggered y }
VAR lo,up : DOTPRECISION;
    i     : INTEGER;
BEGIN
   up:= #( FOR i:= 1 TO UB(y) SUM ( x*y[i] )  );
   lo:= up;
   IF x>0.0 THEN BEGIN
                    lo:= #( lo + x*y[-1] ) ;
                    up:= #( up + x*y[ 0] ) ;
                 END
            ELSE BEGIN
                    lo:= #( lo + x*y[ 0] ) ;
                    up:= #( up + x*y[-1] ) ;
```

```
               END;
   r_ist_MUL:= convert( lo,up );
END;

GLOBAL OPERATOR / ( x : REAL; VAR y : istaggered )
                                       r_ist_DIV : istaggered[-1..stagg_prec];
{ division x / y of a REAL x and an istaggered y }
VAR lo,up : DOTPRECISION;
    i,j   : INTEGER;
    z     : istaggered[-1..stagg_prec];
    ym    : REAL;
    C     : INTERVAL;
BEGIN
   lo  := #( x );
   ym  := #*( FOR i:= 1 TO UB(y) SUM ( y[i] ) + 0.5*y[0] + 0.5*y[-1] );
   z[1]:= x / ym;
   FOR i:= 2 TO stagg_prec DO
   BEGIN
      lo  := #( lo - FOR j:= 1 TO UB(y) SUM ( y[j]*z[i-1] ) );
      z[i]:= #*(lo) / ym;
   END;
   lo:= #( lo - FOR j:= 1 TO UB(y) SUM ( y[j]*z[stagg_prec] ) );
   up:= lo;
   add_i_times_ist( INTVAL(-y[0],-y[-1]), z, lo,up );
   C     := INTVAL( #<(lo),#>(up) ) / ival(y);
   z[-1] := C.INF;
   z[ 0] := C.SUP;
   r_ist_DIV:= z;
END;


{----------------------------------------------------------------------------}
{ operators +, -, *, / for istaggered and REAL :                             }
{----------------------------------------------------------------------------}
GLOBAL OPERATOR  + ( VAR x : istaggered; y : REAL )
                                       ist_r_ADD : istaggered[-1..stagg_prec];
{ addition x + y of an istaggered x and a REAL y }
VAR lo,up : DOTPRECISION;
    i     : INTEGER;
BEGIN
   up:= #( y + FOR i:= 0 TO UB(x) SUM ( x[i] ) );
   lo:= #( up - x[0] + x[-1] );
   ist_r_ADD:= convert( lo,up );
END;

GLOBAL OPERATOR  - ( VAR x : istaggered; y : REAL )
                                       ist_r_SUB : istaggered[-1..stagg_prec];
{ subtraction x - y of an istaggered x and a REAL y }
VAR lo,up : DOTPRECISION;
    i     : INTEGER;
BEGIN
   up:= #( FOR i:= 0 TO UB(x) SUM ( x[i] ) - y );
   lo:= #( up - x[0] + x[-1] );
   ist_r_SUB:= convert( lo,up );
END;

GLOBAL OPERATOR  * ( VAR x : istaggered; y : REAL )
                                       ist_r_MUL : istaggered[-1..stagg_prec];
{ multiplication x * y of an istaggered x and a REAL y }
VAR lo,up : DOTPRECISION;
    i     : INTEGER;
BEGIN
   up:= #( FOR i:= 1 TO UB(x) SUM ( x[i]*y )  );
   lo:= up;
```

```
   IF y>0.0 THEN BEGIN
                    lo:= #( lo + y*x[-1] ) ;
                    up:= #( up + y*x[ 0] ) ;
                 END
            ELSE BEGIN
                    lo:= #( lo + y*x[ 0] ) ;
                    up:= #( up + y*x[-1] ) ;
                 END;
   ist_r_MUL:= convert( lo,up );
END;

GLOBAL OPERATOR / ( VAR x : istaggered; y : REAL )
                                   ist_r_DIV : istaggered[-1..stagg_prec];
{ division x / y of an istaggered x and a REAL y }
VAR lo : DOTPRECISION;
    i  : INTEGER;
    z  : istaggered[-1..stagg_prec];
    C  : INTERVAL;
BEGIN
   lo  := #( FOR i:= 1 TO UB(x) SUM ( x[i] ) );
   z[1]:= #*(lo) / y;
   FOR i:= 2 TO stagg_prec DO
   BEGIN
      lo  := #( lo - y*z[i-1] );
      z[i]:= #*(lo) / y;
   END;
   lo    := #( lo - y*z[stagg_prec] );
   C     := INTVAL( #<(lo+x[-1]),#>(lo+x[0]) ) / y;
   z[-1] := C.INF;
   z[ 0] := C.SUP;
   ist_r_DIV:= z;
END;

{-----------------------------------------------------------------------------}
{ sqrt for istaggered :                                                       }
{-----------------------------------------------------------------------------}
GLOBAL FUNCTION sqrt ( VAR x : istaggered ) : istaggered[-1..stagg_prec];
{ square root of an istaggered x }
VAR lo  : DOTPRECISION;
    i,j : INTEGER;
    z   : istaggered[-1..stagg_prec];
    zi  : REAL;
    C   : INTERVAL;
BEGIN
   IF ival(x) = 0 THEN sqrt:= x ELSE
   BEGIN
      lo  := # ( FOR i:= 1 TO UB(x) SUM ( x[i] ) );
      zi  := SQRT( #*(lo) );
      z[1]:= zi;
      FOR i:= 2 TO stagg_prec DO
      BEGIN
         lo  := #( lo - FOR j:= 1 TO i-2 SUM (zi*z[j] + zi*z[j]) - zi*zi );
         zi  := #*(lo) / (2*z[1]);
         z[i]:= zi;
      END;
      lo    := #( lo - FOR j:= 1 TO stagg_prec-1
                   SUM ( zi*z[j] + zi*z[j] ) - zi*zi );
      z[-1] := 0;
      z[ 0] := 0;
      C     := INTVAL( #<(lo+x[-1]),#>(lo+x[0]) ) / (SQRT(ival(x)) + ival(z));
      z[-1] := C.INF;
      z[ 0] := C.SUP;
      sqrt:= z;
   END;
```

```
END;

BEGIN
   stagg_prec:= 1;                      { set initial staggered length to 1 }
END.
```

6.5 Test Results

In this section we will demonstrate a simple but powerful application of staggered interval arithmetic to discrete dynamic systems. Additionally, in the following Section 6.7 we will give some further references to other applications which can be found in the literature.

The computation of orbits of *dynamic systems* is known to be highly unstable if the system exhibits chaotic behavior. In this case, even for the very simplest systems, ordinary floating-point computations will eventually deliver results which are completely wrong quantitatively. Also ordinary interval arithmetic (i.e. intervals of floating-point numbers) will yield poor enclosures after few iterations and, finally, in most cases the computation will break down because of overflow.

By use of the interval staggered correction format, however, we can compute enclosures of orbits for a considerably longer time with high accuracy. Of course this can also be achieved by means of multi-precision arithmetic simulated in software; once the long accumulator is available in hardware, the staggered arithmetic will fully benefit from the speed of floating-point hardware.

Consider the simple dynamic system as given by the *logistic equation*:

$$x_{n+1} = a \cdot x_n \cdot (1 - x_n)\ , \quad n \ \geq \ 0 \tag{6.13}$$

for some $a \in [0,4]$ and $x_0 \in (0,1)$.

On the computer, we can compute this iteration with (i) ordinary floating-point arithmetic, (ii) ordinary interval arithmetic or with (iii) staggered interval arithmetic. However, for the cases (ii) and (iii) we will first rewrite the right hand side of (6.13) such that it is better suited for the application of interval arithmetic: For narrow intervals it is well known in interval analysis that a tighter interval enclosure can be obtained by using a *mean value form* instead of an interval evaluation of the originally given expression.

The ordinary interval evaluation of a function $f(x)$ over an interval X, denoted as $f(X)$, is obtained via replacing all occurrences of x in f by the interval X and via replacing all operations by the corresponding interval operations. The mean value form is defined by $f_m(X) := f(y) + f'(X)(X - y)$ with some fixed value $y \in X$, e.g., the midpoint. Thus, in the cases (ii) and (iii) we replace the right hand side of (6.13) by its mean value form, i.e. , by

$$\begin{array}{l} X_{n+1} = a \cdot (y_n(1 - y_n) + (1 - 2X_n) \cdot (X_n - y_n)) \\ \quad \text{with} \quad y_n \ \approx \ \text{mid}(X_n) \ = \ \text{midpoint of } X_n \quad . \end{array} \tag{6.14}$$

where X_n is an interval in case (ii) and a staggered interval in case (iii). Rewriting (6.13) as (6.14) does not affect the quality of ordinary floating-point computation, which is still executed using (6.13).

The following PASCAL–XSC program uses the module *istagger* from the previous section to compute orbits for this equation. The print-out lists the approximations x_n obtained by ordinary floating-point evaluations of (6.13) and the enclosures X_n as obtained by ordinary and by staggered interval arithmetic using (6.14).

```
PROGRAM ICHAOS;
USE i_ari,istagger; { Import interval and staggered interval arithmetic }

PROCEDURE MAIN;
VAR x,xm : istaggered[-1..stagg_prec];
    y,ym : INTERVAL;
    a,x0 : REAL;
    n,m  : INTEGER;
BEGIN
   WRITELN('Computation of the logistic equation');
   WRITELN('  x(n+1) = a * x(n) * ( 1-x(n) )');
   WRITELN;
   WRITE('enter parameter      a = '); READ(a);
   WRITE('enter initial value x0 = '); READ(x0);
   WRITE('number of iterations m = '); READ(m);
   y:= x0;    { initial value for interval arithmetic           }
   x:= x0;    { initial value for staggered interval arithmetic }
   FOR n:= 1 TO m DO
   BEGIN
      x0:= a*x0*(1-x0);      {  Compute real approximation      }
      {-----------------------------------------------------}
      { Compute with interval and staggered interval arithmetic }
      { of length stagg_prec. Use mean value form for x(n+1) :  }
      {-----------------------------------------------------}
      ym:= mid(y);   y := a*( ym*(1-ym) + (1-2*y)*(y-ym) );
      xm:= mid(x);   x := a*( xm*(1-xm) + (1-2*x)*(x-xm) );
      WRITELN(n, ' : ', x0, y, IVAL(x) );
   END;
END;
BEGIN
   WRITE('enter staggered length : '); READ(stagg_prec);
   IF stagg_prec>0 THEN MAIN;
END.
```

For the input data $a = 3.75$, $x_0 = 0.5$ and $n = 500$ iterations we get the following results:

In order to demonstrate the effect of the choice of (6.13) or (6.14), in the following Table 6.2 we list the maximum number n_{max} of iterations which can be performed with staggered interval arithmetic by use of (6.13) or (6.14) and with different choices of the staggered length *stagg_prec*. For larger values of n overflow occurs and the program is aborted (all computations were carried out on the basis of the IEEE double floating-point format).

In all cases here, the accuracy obtained when rounding the results to S is very high (i.e. the maximum accuracy). The accuracy decreases only immediately before reaching n_{max}. Pure floating-point computations always yield totally incorrect results after more than about 100 iterations.

n	*floating-point*	*ordinary interval*	*staggered interval*
1	0.9375000000000000	0.9375000000000000	0.9375000000000000
10	0.6453672908309288	$0.6453672908309_{231}^{382}$	$0.645367290830930_{2}^{4}$
20	0.8259709787108499	$0.82597097871_{03328}^{11930}$	$0.825970978710775_{4}^{6}$
30	0.7180965684239893	$0.718096568_{3876004}^{4481460}$	$0.718096568418754_{6}^{8}$
40	0.4163493160568014	$0.4163493_{118996043}^{223190691}$	$0.416349316957635_{6}^{7}$
50	0.3604395431283658	$0.3604_{391241304220}^{401742922880}$	$0.360439633921698_{5}^{7}$
60	0.7990957294909673	$0.799_{0303938934906}^{1391103575510}$	$0.799086334308309_{4}^{6}$
70	0.4520542523856749	$0.45_{12732497659017}^{31654999449893}$	$0.452195299859730_{1}^{2}$
80	0.8492167613301459	$0._{7677390706389728}^{9469520868573489}$	$0.856177996629336_{4}^{6}$
90	0.8481094253763446	[0,1]	$0.73991374860737_{09}^{11}$
100	0.8358933994881687	[0,1]	$0.88829399228403_{48}^{50}$
150	0.8504150611290033	[0,1]	$0.702820413487813_{5}^{7}$
200	0.9131976055921124	[0,1]	$0.82355732113045_{69}^{71}$
250	0.9310785715888734	[0,1]	$0.219734861549818_{1}^{2}$
300	0.3198944553532391	[0,1]	$0.496432049973525_{6}^{8}$
350	0.9287366136360928	[0,1]	$0.859325800193209_{0}^{2}$
400	0.2202363784969592	[0,1]	$0.69393944643472_{19}^{21}$
450	0.9345460787353592	[0,1]	$0.61598678450720_{11}^{31}$
500	0.9245450657841405	[0,1]	$0.2767538_{665021436}^{922571261}$

Table 6.1: logistic equation for $a = 3.75$, $x_0 = 0.5$; last column: staggered length *stagg_prec*= 5

6.6 Exercises

6.7 Notes and References

Elementary functions can be implemented for staggered intervals without difficulties. The algorithms for such implementations may already make use of the staggered operations presented here. As an example, $\exp(x)$ can be implemented by use of the Taylor series which is computed in staggered interval arithmetic, while the remainder term is just an ordinary interval to be added to the interval part of the staggered result. For the inverse functions, it is often very easy to apply an interval Newton's method using staggered interval arithmetic. As an example, $z = \ln x$ can be obtained as the zero of the function $f(z) = x - exp(z)$. Here, only $exp(z)$ and staggered interval arithmetic are needed. A Newton method of this kind can even be implemented quite efficiently making use of the quadratic convergence property: we start the iteration with a small staggered length n_z doubling this length after each iteration step. The overall computational cost will then roughly be only twice as much as the cost for one iteration employing the maximum staggered length that is used.

stagg_prec	$n_{\max}$ with (6.13)	$n_{\max}$ with (6.14)
0	35	90
1	69	169
2	99	277
4	152	464
6	208	636
8	262	846
10	315	979
14	424	1303
18	532	1679

Table 6.2: max. iteration count with staggered arithmetic

A different approach to compute results in a staggered correction format avoids a separate implementation of each basic arithmetic operation; rather, the algorithm is rearranged such that it produces its result already in a staggered correction format. Algorithms for linear problems can often be easily transformed in such a way (see e.g. [285]). In fact many of the algorithms in this book compute their results in a format which resembles more or less a staggered correction interval form, see e.g. the algorithms in Chapter 2 and Chapter 3.

The staggered correction format was introduced by Rump, [280], [285], and Böhm, [60], [61], such that elementary operations are not used explicitly. Stetter and Auzinger, [306], [27], proposed the use of this format in a more general, operation-wise way and also coined its name. Klotz, [180], discussed several matrix factorizations using staggered correction arithmetic; since then, staggered correction methods have also been applied to other problems; e.g. the evaluation of multivariate polynomials, [230], the computation of all eigenvalues of symmetric matrices, [231], and the computation of the matrix exponential function, [45], [46].

As a last application we mention *ordinary initial value problems*. The algorithm from [229] which computes guaranteed continuous bounds for the solutions of nonlinear initial value problems, gives its results in the form $x(t) \in \tilde{x}(t) + R(t)$ where $\tilde{x}(t)$ is an approximation of the solution and $R(t)$ is an interval enclosing the global error. Here, it is possible to apply staggered arithmetic in the computation of $\tilde{x}(t)$, however to retain ordinary interval arithmetic in the computation of $R(t)$, which is the most expensive part of the algorithm. Thus, the accuracy of the results can be increased, mainly by increasing the accuracy of the approximation. The increase of the cost for the computation of the global error $R(t)$ is almost negligible.

Chapter 7

Multiple-Precision Arithmetic Using Integer Operations

Modules for multiple-precision real [201] and interval arithmetic [200, 203] have been developed in PASCAL-XSC. These modules provide a lot of predefined operations and standard functions for the multiple-precision data types. Due to the concept of overloading of functions and the operator concept of PASCAL-XSC a mathematical notation of expressions also for the multiple-precision data types is possible, which makes the modules applicable in an easy and natural way. Programs become clear and readable.

Both modules described here are under further developement.

In the runtime library of PASCAL-XSC [81], a multiple-precision arithmetic is available [83]. The base used for the representation of mantissa digits is $B = 2^{32}$, i.e., each mantissa digit occupies 32 bits and is implemented by an unsigned 32-bit integer value. Additionaly there are an exponent field as well as some other flags which, for example, indicate that a multiple-precision value is (a) zero, (b) negative, (c) temporary, or (d) exact. The number of mantissa digits that may be used to represent one multiple-precision value is $l_{\max} := 2^{32} - 1$. Such a multiple-precision number needs about $2^{32} \cdot 4 \approx 17 \cdot 10^9$ bytes of storage for its representation. Thus on many machines a further limitation will come from the restricted capacity of the memory of the employed machine. A positive multiple-precision number $x \neq 0$ can be written in the form

$$x = \sum_{n=0}^{l_{\max}-1} m_n B^{ex-n} \ , \quad m_0 \neq 0, \ m_n \in \{0, 1, \ldots, 2^{32} - 1\} \tag{7.1}$$

The exponent ex ranges from $e_{\min} := -2^{31}$ to $e_{\max} := 2^{31} - 1$. For example, the multiple-precision number 0.1_B is equal to $1 \cdot B^{-1} = 1/2^{32} = 0.000\ldots01_2 = 2.3283064\ldots_{10} \cdot 10^{-10}$. Equivalently $0.11|2531|93_B \cdot B^2 = 11 \cdot B^1 + 2531 \cdot B^0 + 93 \cdot B^{-1} = 11 \cdot 2^{32} + 2531 + 93 \cdot 2^{-32}$. Here we separate individual mantissa digits of the multiple-precision number by the symbol $|$.

The required number of mantissa digits (with respect to base 2^{32}) of a resulting value may be set separately for each operation. All operations consider their arguments to full-length independently from the required length of the result mantissa.

There are numerous elementary mathematical functions such as log, exp, sin, cos, arcsin, ... for arguments of the multiple-precision data type. The values of these functions are guaranteed to be accurate to at least two units in the last mantissa digit of the results. Again the number of required mantissa digits may be chosen arbitrarily.

The PASCAL-XSC module for real multi-precision computations is named mp_ari. The multiple-precision data type is called mpreal. The standard procedure to set the precision is setprec(n). The integer value n specifies the number of required mantissa digits of the results of all subsequent operations. The integer standard function getprec returns the current precision of the multiple-precision arithmetic. Function getprec has no arguments. The default precision setting for the arithmetical operations is 3. This means that the result of a multi-precision operation is chopped after 3 mantissa digits. The default setting especially guarantees, that each IEEE double number (53 mantissa bits) can be converted without conversion error into a multi-precision number.

The module for multi-precision interval arithmetic is called mpi_ari. The multi-precision interval data type is named mpinterval. The precision setting is achieved using the same procedure setprec(n) already mentioned. The result of an elementary mathematical function for this data type is a multiple-precision interval. The resulting interval is a superset of the range of the function over the input interval. The bounds are accurate to at least one unit in the last place with respect to the actual precision setting. Again the precision setting may be changed at any point of the program.

The multiple-precision operations $+$, $-$, $*$, $/$ as well as the elementary mathematical functions are called in the usual way. Here, the operator concept of PASCAL-XSC and the concept of overloading of function names is extensively used. This leads to clear and concise programs which are easy to read and debug.

7.1 Multi-Precision Real Arithmetic

We list the entries of the PASCAL-XSC module mp_ari for the multi-precision real arithmetic. It includes the definition of the new global data type mpreal for real multi-precision numbers.

The following lines of code are extracted from the PASCAL-XSC source mp_ari.p. The simple PASCAL-XSC program which performs this task uses operators for strings and is as follows:

```
PROGRAM GlobOut;
{--------------------------------------------------------------------}
{ Purpose: List global functions, operators, and and procedures of   }
{          a PSCAL-XSC module.                                       }
{--------------------------------------------------------------------}
VAR
  InFile  : text;
  FileName: string;
  row     : string;
  NumberOfRows: integer;
BEGIN
  Filename:= 'mp_ari.p';
  writeln('File to be processed: ' + FileName);
  reset(InFile, FileName);
  NumberOfRows:= 0;
  WHILE NOT eof(InFile) DO BEGIN
```

```
    NumberOfRows:= NumberOfRows+1;
    readln(InFile, row);
    IF ('global' IN row) OR ('{=' IN row) THEN writeln(row);
  END;
  writeln('Number of rows processed: ', NumberOfRows);
END.
{----------------------------------------------------------------------}
```

Only lines beginning with `global` or beginning with `{=` are listed. For the complete source code of the real multi-precision module see the file `mp_ari.p`.

```
FILE TO be processed: mp_ari.p
{============================================================================}
{=====  mp_ari  ==========  Multi-Precision Arithmetic  ===================}
{=======================  =     =        ===         ===================}
{============================================================================}
{=  Version :  1.1                                                          }
{=  Date    :  05/29/91  ( first implementation )                           }
{=             03/21/95  ( last change )                                    }
{============================================================================}
{============================================================================}
{==========  Datatypes  =====================================================}
{============================================================================}
GLOBAL TYPE  mpreal = ↑integer;  { Real multiple-precision data type }
GLOBAL TYPE  mpmode = GLOBAL (InitFunc, TempFunc, InitOp, TempOp);
                      {== Needed for allocation of the result variable ==}
                      {== in an operator or function definition.       ==}
{============================================================================}
{==========  Memory Management  =============================================}
{============================================================================}
GLOBAL PROCEDURE mpinit( VAR x : mpreal );    { Initialize a mpreal variable }
GLOBAL PROCEDURE mpvlcp( VAR x : mpreal );    { Get a local copy of a mpreal }
GLOBAL PROCEDURE mpfree( VAR x : mpreal );    { Free a mpreal variable       }
GLOBAL PROCEDURE mptemp( VAR x : mpreal );    { Mark a mpreal variable to be }
GLOBAL PROCEDURE mputmp( VAR x : mpreal );    { Reset the temporary flag of  }
GLOBAL FUNCTION mpttst( VAR x : mpreal ) : boolean; { Test a mpreal variable }
{============================================================================}
{==========  I/O - Routines  ================================================}
{============================================================================}
GLOBAL PROCEDURE read( VAR t : text; VAR x : mpreal );
GLOBAL PROCEDURE read( VAR t : text; VAR x : mpreal; i : integer );
GLOBAL PROCEDURE write( VAR t : text; x : mpreal );
GLOBAL PROCEDURE write( VAR t : text; x : mpreal; i : integer );
GLOBAL PROCEDURE write( VAR t : text; x : mpreal; i, j : integer );
GLOBAL PROCEDURE write( VAR t : text; x : mpreal; i, j, k : integer );
GLOBAL PROCEDURE writehex( VAR t : text; x : mpreal; mode : char );
{============================================================================}
{==========  Exception Handling  ============================================}
{============================================================================}
{============================================================================}
{==========  Logical Operators: mpreal <--> mpreal  ======================}
{============================================================================}
GLOBAL OPERATOR =  ( x, y : mpreal ) res : boolean;
GLOBAL OPERATOR >= ( x, y : mpreal ) res : boolean;
GLOBAL OPERATOR >  ( x, y : mpreal ) res : boolean;
GLOBAL OPERATOR <= ( x, y : mpreal ) res : boolean;
GLOBAL OPERATOR <  ( x, y : mpreal ) res : boolean;
GLOBAL OPERATOR <> ( x, y : mpreal ) res : boolean;
{============================================================================}
{====================  Type Conversion (Local)  ==========================}
{============================================================================}
{============================================================================}
{==========  Special Assignment Operators  =================================}
```

```
{==========================================================================}
GLOBAL OPERATOR := ( VAR x : mpreal; mode : mpmode );
GLOBAL OPERATOR := ( VAR x : mpreal; mode : boolean );
{==========================================================================}
{==========  Assignment Operators  ========================================}
{==========================================================================}
GLOBAL OPERATOR := ( VAR x : mpreal; y : mpreal );        { mpreal := mpreal }
GLOBAL OPERATOR := ( VAR x : mpreal; r : real );            { mpreal := real }
GLOBAL OPERATOR := ( VAR x : mpreal; i : integer );      { mpreal := integer }
GLOBAL OPERATOR := ( VAR r : real; x : mpreal );            { real := mpreal }
{==========================================================================}
{==========  Functions for Type Coercions  ================================}
{==========================================================================}
GLOBAL FUNCTION _mpreal( r : real ) : mpreal;                { real coercion }
GLOBAL FUNCTION _mpreal( r : real; rnd : integer ) : mpreal; { real-coercion }
GLOBAL FUNCTION _mpreal( i : integer ) : mpreal;          { integer-coercion }
GLOBAL FUNCTION _mpreal( i : integer; rnd : integer ) : mpreal;
GLOBAL FUNCTION _real( x : mpreal ) : real;              { real-coercion for }
GLOBAL FUNCTION _real( x : mpreal; rnd : integer ) : real;
{==========================================================================}
{==========  Precision Control  ===========================================}
{==========================================================================}
GLOBAL PROCEDURE setprec( i : integer );            { Set actual precision }
GLOBAL FUNCTION getprec : integer;                  { Get actual precision }
{==========================================================================}
{==========  Logical Operators: mpreal <--> real     ======================}
{==========                     real   <--> mpreal   ======================}
{==========================================================================}
GLOBAL OPERATOR =  ( r : real; x : mpreal ) eq  : boolean;
GLOBAL OPERATOR =  ( x : mpreal; r : real ) eq  : boolean;
GLOBAL OPERATOR <> ( r : real; x : mpreal ) neq : boolean;
GLOBAL OPERATOR <> ( x : mpreal; r : real ) neq : boolean;
GLOBAL OPERATOR >  ( r : real; x : mpreal ) gt  : boolean;
GLOBAL OPERATOR >  ( x : mpreal; r : real ) gt  : boolean;
GLOBAL OPERATOR >= ( r : real; x : mpreal ) ge  : boolean;
GLOBAL OPERATOR >= ( x : mpreal; r : real ) ge  : boolean;
GLOBAL OPERATOR <  ( r : real; x : mpreal ) lt  : boolean;
GLOBAL OPERATOR <  ( x : mpreal; r : real ) lt  : boolean;
GLOBAL OPERATOR <= ( r : real; x : mpreal ) le  : boolean;
GLOBAL OPERATOR <= ( x : mpreal; r : real ) le  : boolean;
{==========================================================================}
{==========  Arithmetic Operators: mpreal <--> mpreal  ====================}
{==========================================================================}
GLOBAL OPERATOR +  ( x : mpreal ) mplus     : mpreal;
GLOBAL OPERATOR -  ( x : mpreal ) mminus    : mpreal;
GLOBAL OPERATOR +  ( x, y : mpreal ) add    : mpreal;
GLOBAL OPERATOR +> ( x, y : mpreal ) addu   : mpreal;
GLOBAL OPERATOR +< ( x, y : mpreal ) addd   : mpreal;
GLOBAL OPERATOR -  ( x, y : mpreal ) sub    : mpreal;
GLOBAL OPERATOR -> ( x, y : mpreal ) subu   : mpreal;
GLOBAL OPERATOR -< ( x, y : mpreal ) subd   : mpreal;
GLOBAL OPERATOR *  ( x, y : mpreal ) mul    : mpreal;
GLOBAL OPERATOR *> ( x, y : mpreal ) mulu   : mpreal;
GLOBAL OPERATOR *< ( x, y : mpreal ) muld   : mpreal;
GLOBAL OPERATOR /  ( x, y : mpreal ) frac   : mpreal;
GLOBAL OPERATOR /> ( x, y : mpreal ) fracu  : mpreal;
GLOBAL OPERATOR /< ( x, y : mpreal ) fracd  : mpreal;
{==========================================================================}
{==========  Arithmetic Operators: real   <--> mpreal  ====================}
{==========                        mpreal <--> real    ====================}
{==========================================================================}
GLOBAL OPERATOR +  ( r : real; x : mpreal ) add   : mpreal;
GLOBAL OPERATOR +> ( r : real; x : mpreal ) addu  : mpreal;
GLOBAL OPERATOR +< ( r : real; x : mpreal ) addd  : mpreal;
```

```
GLOBAL OPERATOR +  ( x : mpreal; r : real ) add   : mpreal;
GLOBAL OPERATOR +> ( x : mpreal; r : real ) addu  : mpreal;
GLOBAL OPERATOR +< ( x : mpreal; r : real ) addd  : mpreal;
GLOBAL OPERATOR -  ( r : real; x : mpreal ) sub   : mpreal;
GLOBAL OPERATOR -> ( r : real; x : mpreal ) subu  : mpreal;
GLOBAL OPERATOR -< ( r : real; x : mpreal ) subd  : mpreal;
GLOBAL OPERATOR -  ( x : mpreal; r : real ) sub   : mpreal;
GLOBAL OPERATOR -> ( x : mpreal; r : real ) subu  : mpreal;
GLOBAL OPERATOR -< ( x : mpreal; r : real ) subd  : mpreal;
GLOBAL OPERATOR *  ( r : real; x : mpreal ) mul   : mpreal;
GLOBAL OPERATOR *> ( r : real; x : mpreal ) mulu  : mpreal;
GLOBAL OPERATOR *< ( r : real; x : mpreal ) muld  : mpreal;
GLOBAL OPERATOR *  ( x : mpreal; r : real ) mul   : mpreal;
GLOBAL OPERATOR *> ( x : mpreal; r : real ) mulu  : mpreal;
GLOBAL OPERATOR *< ( x : mpreal; r : real ) muld  : mpreal;
GLOBAL OPERATOR /  ( r : real; x : mpreal ) frac  : mpreal;
GLOBAL OPERATOR /> ( r : real; x : mpreal ) fracu : mpreal;
GLOBAL OPERATOR /< ( r : real; x : mpreal ) fracd : mpreal;
GLOBAL OPERATOR /  ( x : mpreal; r : real ) frac  : mpreal;
GLOBAL OPERATOR /> ( x : mpreal; r : real ) fracu : mpreal;
GLOBAL OPERATOR /< ( x : mpreal; r : real ) fracd : mpreal;
{============================================================================}
{==========   Arithmetic Operators: integer <--> mpreal    ==================}
{==========                         mpreal  <--> integer   ==================}
{============================================================================}
GLOBAL OPERATOR *  ( i : integer; x : mpreal ) mul   : mpreal;
GLOBAL OPERATOR *> ( i : integer; x : mpreal ) mulu  : mpreal;
GLOBAL OPERATOR *< ( i : integer; x : mpreal ) muld  : mpreal;
GLOBAL OPERATOR *  ( x : mpreal; i : integer ) mul   : mpreal;
GLOBAL OPERATOR *> ( x : mpreal; i : integer ) mulu  : mpreal;
GLOBAL OPERATOR *< ( x : mpreal; i : integer ) muld  : mpreal;
GLOBAL OPERATOR /  ( i : integer; x : mpreal ) frac  : mpreal;
GLOBAL OPERATOR /> ( i : integer; x : mpreal ) fracu : mpreal;
GLOBAL OPERATOR /< ( i : integer; x : mpreal ) fracd : mpreal;
GLOBAL OPERATOR /  ( x : mpreal; i : integer ) frac  : mpreal;
GLOBAL OPERATOR /> ( x : mpreal; i : integer ) fracu : mpreal;
GLOBAL OPERATOR /< ( x : mpreal; i : integer ) fracd : mpreal;
{============================================================================}
{==========   Standard Functions   ==========================================}
{============================================================================}
GLOBAL FUNCTION mpval( s : string ) : mpreal;        { No rounding parameter }
GLOBAL FUNCTION mpval( s : string; rnd : integer ) : mpreal;
GLOBAL FUNCTION comp( mant : mpreal; expo : integer ) : mpreal;
GLOBAL FUNCTION expo( x : mpreal ) : integer;
GLOBAL FUNCTION mant( x : mpreal ) : mpreal;
GLOBAL FUNCTION pred( x : mpreal ) : mpreal;
GLOBAL FUNCTION succ( x : mpreal ) : mpreal;
GLOBAL FUNCTION sign( x : mpreal ) : integer;
GLOBAL FUNCTION trunc( x : mpreal ) : integer;
GLOBAL FUNCTION round( x : mpreal ) : integer;
GLOBAL FUNCTION abs( x : mpreal ) : mpreal;
GLOBAL FUNCTION sqr( x : mpreal ) : mpreal;
GLOBAL FUNCTION sqrt( x : mpreal ) : mpreal;
GLOBAL FUNCTION ln( x : mpreal ) : mpreal;
GLOBAL FUNCTION loga( x, a : mpreal ) : mpreal;
GLOBAL FUNCTION log2( x: mpreal ) : mpreal;
GLOBAL FUNCTION log10( x: mpreal ) : mpreal;
GLOBAL FUNCTION exp( x : mpreal ) : mpreal;
GLOBAL FUNCTION power( x, y : mpreal ) : mpreal;
GLOBAL FUNCTION exp2( x : mpreal ) : mpreal;
GLOBAL FUNCTION exp10( x : mpreal ) : mpreal;
GLOBAL FUNCTION sin( x : mpreal ) : mpreal;
GLOBAL FUNCTION cos( x : mpreal ) : mpreal;
GLOBAL FUNCTION tan( x : mpreal ) : mpreal;
```

```
GLOBAL FUNCTION cot( x : mpreal ) : mpreal;
GLOBAL FUNCTION arcsin( x : mpreal ) : mpreal;
GLOBAL FUNCTION arccos( x : mpreal ) : mpreal;
GLOBAL FUNCTION arctan( x : mpreal ) : mpreal;
GLOBAL FUNCTION arccot( x : mpreal ) : mpreal;
GLOBAL FUNCTION arctan2( x, y : mpreal ) : mpreal;
GLOBAL FUNCTION sinh( x : mpreal ) : mpreal;
GLOBAL FUNCTION cosh( x : mpreal ) : mpreal;
GLOBAL FUNCTION tanh( x : mpreal ) : mpreal;
GLOBAL FUNCTION coth( x : mpreal ) : mpreal;
GLOBAL FUNCTION arsinh( x : mpreal ) : mpreal;
GLOBAL FUNCTION arcosh( x : mpreal ) : mpreal;
GLOBAL FUNCTION artanh( x : mpreal ) : mpreal;
GLOBAL FUNCTION arcoth( x : mpreal ) : mpreal;
{==============================================================================}
{==========   Miscellaneous Functions   =====================================}
{==============================================================================}
GLOBAL FUNCTION mant_length( x : mpreal ) : integer;   { Number of mp-digits }
GLOBAL FUNCTION copy( x : mpreal ) : mpreal;         { Exact copy, ignoring }
GLOBAL FUNCTION trunc_mp( x : mpreal ) : mpreal;  { Integer part of a mpreal }
GLOBAL FUNCTION odd_mp( x : mpreal) : boolean;     { Test for a mpreal to be }
GLOBAL FUNCTION min( x, y : mpreal ) : mpreal;      { Minimum of two mpreals }
GLOBAL FUNCTION max( x, y : mpreal ) : mpreal;      { Maximum of two mpreals }
{==============================================================================}
{==========   Module Initalization Part   ===================================}
{==============================================================================}
{===================  Last line of module mp_ari: 1163  ====================}
Number OF rows processed: 1163
```

As may be seen by the preceding listing a large variety of common operators and elementary mathematical functions are realized in this module for multi-precision computations. The frequently used standard function `expo(mp)` gives the exponent of `mp` with respect to base 2 and `mant(mp)` gives the corresponding mantissa of its mpreal argument `mp` normalized to $0.5 \leq$ `mant(mp)` < 1.0. For the composition function `comp` it holds `comp(mant(mp), expo(mp))` $=$ `mp`.

7.2 Multi-Precision Interval Arithmetic

We now list the entries of the PASCAL-XSC module `mpi_ari` for the multi-precision real interval arithmetic. It includes the definition of the new global data type `mpinterval` for multi-precision intervals. Again only lines beginning with `global` or beginning with `{=` are printed. For the complete source code for the multi-precision interval module see the file `mpi_ari.p`.

```
FILE TO be processed: mpi_ari.p
{==============================================================================}
{===  mpi_ari  ======   Multi-Precision Interval Arithmetic   ===============}
{==================   =     =         =        ===          ===============}
{==============================================================================}
{=   Version :  1.1                                                         }
{=   Date    :  10/07/91  ( first implementation )                          }
{=              09/21/95  ( last change )                                   }
{==============================================================================}
USE GLOBAL
{==============================================================================}
{==========   Datatypes and Variables   =====================================}
```

```
{============================================================================}
GLOBAL TYPE  mpinterval  =  GLOBAL RECORD   inf,  sup: mpreal;  END;
{============================================================================}
{==========  Memory Management   ============================================}
{============================================================================}
GLOBAL PROCEDURE mpinit( VAR x : mpinterval );     { Initialize a mpinterval }
GLOBAL PROCEDURE mpvlcp ( VAR x : mpinterval );    { Get a local copy of a   }
GLOBAL PROCEDURE mpfree ( VAR x : mpinterval );    { Free a mpinterval       }
{============================================================================}
{==========  I/O - Routines  ================================================}
{============================================================================}
GLOBAL PROCEDURE read( VAR t : text; VAR x : mpinterval );   { Input without }
GLOBAL PROCEDURE write( VAR t : text; x : mpinterval );
{============================================================================}
{==========  Exception Handling  ============================================}
{============================================================================}
{============================================================================}
{==========  Special Assignment Operators  ==================================}
{============================================================================}
{= A special assignment operator is used to initialize the result variable of}
{= a function (InitFunc) or operator (InitOp) or to mark the result variable }
{= to be temporary (TempFunc, TempOp). 'InitFunc' and 'InitOp' are synonyms. }
{= 'TempFunc' and 'TempOp' are also synonyms. For compatibility to older     }
{= versions of the multi-precision modules there also exists an assignment   }
{= operator of type boolean. The value 'true' is equivalent to 'InitFunc' and}
{= 'InitOp', the value 'false' is equivalent to 'TempFunc' and 'TempOp'. For }
{= definition of type 'mpmode' see module mp_ari.                            }
{=---------------------------------------------------------------------------}
GLOBAL OPERATOR := ( VAR x : mpinterval; mode : mpmode );
GLOBAL OPERATOR := ( VAR x : mpinterval; mode : boolean );
{============================================================================}
{==========  Assignment Operators  ==========================================}
{============================================================================}
GLOBAL OPERATOR := ( VAR x : mpinterval;         { mpinterval := mpinterval }
GLOBAL OPERATOR := ( VAR x : mpinterval;           { mpinterval := interval }
GLOBAL OPERATOR := ( VAR x : mpinterval;             { mpinterval := mpreal }
GLOBAL OPERATOR := ( VAR x : mpinterval;               { mpinterval := real }
GLOBAL OPERATOR := ( VAR x : interval;             { interval := mpinterval }
{============================================================================}
{==========  Functions for Type Coercions  ==================================}
{============================================================================}
GLOBAL FUNCTION _mpinterval( x : mpreal ) : mpinterval;    { mpreal-coercion }
GLOBAL FUNCTION _mpinterval( x : interval ) : mpinterval;  { interval-coerc. }
GLOBAL FUNCTION _mpinterval( r : real ) : mpinterval;        { real-coercion }
{============================================================================}
{==========  Logical Operators: mpinterval <--> mpinterval   ================}
{============================================================================}
GLOBAL OPERATOR = ( x, y : mpinterval ) eq : boolean;             { Equality }
GLOBAL OPERATOR <> ( x, y : mpinterval ) neq : boolean;         { Unequality }
GLOBAL OPERATOR <= ( x, y : mpinterval ) sub : boolean;             { Subset }
GLOBAL OPERATOR < ( x, y : mpinterval ) psub : boolean;      { Proper subset }
GLOBAL OPERATOR >= ( x, y : mpinterval ) sup : boolean;           { Superset }
GLOBAL OPERATOR > ( x, y : mpinterval ) psup : boolean;   { Proper Superset }
GLOBAL OPERATOR IN ( x, y : mpinterval ) cont : boolean; { Contained-in test }
GLOBAL OPERATOR >< ( x, y : mpinterval ) disj : boolean;    { Disjointedness }
GLOBAL FUNCTION point( x : mpinterval ) : boolean;     { Point interval test }
{============================================================================}
{==========  Logical Operators: mpreal     <--> mpinterval   ================}
{==========                     mpinterval <--> mpreal       ================}
{============================================================================}
GLOBAL OPERATOR = ( x : mpinterval; a : mpreal ) eq : boolean;
GLOBAL OPERATOR = ( a : mpreal; x : mpinterval ) eq : boolean;
GLOBAL OPERATOR IN ( a : mpreal; x : mpinterval ) cont : boolean;
{============================================================================}
```

```
{==========  Logical Operators: real      <--> mpinterval   ===============}
{==========                     mpinterval <--> real        ===============}
{=========================================================================}
GLOBAL OPERATOR = ( x : mpinterval; r : real ) eq : boolean;
GLOBAL OPERATOR = ( r : real; x : mpinterval ) eq : boolean;
GLOBAL OPERATOR IN ( r : real; x : mpinterval ) cont : boolean;
{=========================================================================}
{==========  Logical Operators: interval   <--> mpinterval   ===============}
{==========                     mpinterval <--> interval    ===============}
{=========================================================================}
GLOBAL OPERATOR = ( x : mpinterval; a : interval ) eq : boolean;
GLOBAL OPERATOR = ( a : interval; x : mpinterval ) eq : boolean;
GLOBAL OPERATOR IN ( a : interval; x : mpinterval ) cont : boolean;
{=========================================================================}
{==========  Arithmetic Operators: mpinterval <--> mpinterval   ============}
{=========================================================================}
GLOBAL OPERATOR + ( x : mpinterval ) mplus : mpinterval;
GLOBAL OPERATOR - ( x : mpinterval ) mminus : mpinterval;
GLOBAL OPERATOR + ( x, y : mpinterval ) add : mpinterval;
GLOBAL OPERATOR - ( x, y : mpinterval ) sub : mpinterval;
GLOBAL OPERATOR * ( x, y : mpinterval ) mul : mpinterval;
GLOBAL OPERATOR / ( x, y : mpinterval ) frac : mpinterval;
{=========================================================================}
{==========  Arithmetic Operators: mpreal     <--> mpinterval   ============}
{==========                        mpinterval <--> mpreal       ============}
{=========================================================================}
GLOBAL OPERATOR + ( a : mpreal; x : mpinterval ) add : mpinterval;
GLOBAL OPERATOR + ( x : mpinterval ; a : mpreal ) add : mpinterval;
GLOBAL OPERATOR - ( a : mpreal; x : mpinterval ) sub : mpinterval;
GLOBAL OPERATOR - ( x : mpinterval ; a : mpreal ) sub : mpinterval;
GLOBAL OPERATOR * ( a : mpreal; x : mpinterval ) mul : mpinterval;
GLOBAL OPERATOR * ( x : mpinterval ; a : mpreal ) mul : mpinterval;
GLOBAL OPERATOR / ( a : mpreal; x : mpinterval ) frac : mpinterval;
GLOBAL OPERATOR / ( x : mpinterval ; a : mpreal ) frac : mpinterval;
{=========================================================================}
{==========  Arithmetic Operators: real       <--> mpinterval   ============}
{==========                        mpinterval <--> real         ============}
{=========================================================================}
GLOBAL OPERATOR + ( r : real; x : mpinterval ) add : mpinterval;
GLOBAL OPERATOR + ( x : mpinterval; r : real ) add : mpinterval;
GLOBAL OPERATOR - ( r : real; x : mpinterval ) sub : mpinterval;
GLOBAL OPERATOR - ( x : mpinterval; r : real ) sub : mpinterval;
GLOBAL OPERATOR * ( r : real; x : mpinterval ) mul : mpinterval;
GLOBAL OPERATOR * ( x : mpinterval; r : real ) mul : mpinterval;
GLOBAL OPERATOR / ( r : real; x : mpinterval ) frac : mpinterval;
GLOBAL OPERATOR / ( x : mpinterval; r : real ) frac : mpinterval;
{=========================================================================}
{==========  Arithmetic Operators: interval   <--> mpinterval   ============}
{==========                        mpinterval <--> interval     ============}
{=========================================================================}
GLOBAL OPERATOR + ( a : interval; x : mpinterval ) add : mpinterval;
GLOBAL OPERATOR + ( x : mpinterval ; a : interval ) add : mpinterval;
GLOBAL OPERATOR - ( a : interval; x : mpinterval ) sub : mpinterval;
GLOBAL OPERATOR - ( x : mpinterval ; a : interval ) sub : mpinterval;
GLOBAL OPERATOR * ( a : interval; x : mpinterval ) mul : mpinterval;
GLOBAL OPERATOR * ( x : mpinterval ; a : interval ) mul : mpinterval;
GLOBAL OPERATOR / ( a : interval; x : mpinterval ) frac : mpinterval;
GLOBAL OPERATOR / ( x : mpinterval ; a : interval ) frac : mpinterval;
{=========================================================================}
{==========  Lattice Operators  ==========================================}
{=========================================================================}
GLOBAL OPERATOR +* ( x, y : mpinterval ) hull : mpinterval;    { Convex hull }
GLOBAL OPERATOR ** ( x, y : mpinterval ) intsec : mpinterval; { Intersection }
{=========================================================================}
```

```
{==========   Access and Transfer Functions   ===============================}
{============================================================================}
GLOBAL FUNCTION inf( x : mpinterval ) : mpreal;
GLOBAL FUNCTION sup( x : mpinterval ) : mpreal;
GLOBAL FUNCTION intval( lb, ub : mpreal) : mpinterval;
{============================================================================}
{==========   Utilities   ===================================================}
{============================================================================}
GLOBAL PROCEDURE iround( value: mpreal; VAR lb, ub : mpreal );
GLOBAL FUNCTION is_integer( x: mpinterval ): boolean; { Check for an integer }
{============================================================================}
{==========   Standard Functions   ==========================================}
{============================================================================}
GLOBAL FUNCTION arctan( x : mpinterval ) : mpinterval;
GLOBAL FUNCTION mpival( s : string ) : mpinterval;
GLOBAL FUNCTION mid( x : mpinterval ) : mpreal;          { Interval midpoint }
GLOBAL FUNCTION diam ( x : mpinterval ) : real;     { Real interval diameter }
GLOBAL FUNCTION blow( x : mpinterval ) : mpinterval;           { Blow 2 ULPs }
GLOBAL FUNCTION abs( x : mpinterval ) : mpinterval;         { Absolute value }
GLOBAL FUNCTION sqr( x : mpinterval ) : mpinterval;        { Square function }
GLOBAL FUNCTION sqrt( x : mpinterval ) : mpinterval;           { Square root }
GLOBAL FUNCTION ln( x : mpinterval ) : mpinterval;       { Natural logarithm }
GLOBAL FUNCTION log2( x : mpinterval ) : mpinterval;     { Logarithm, base 2 }
GLOBAL FUNCTION log10( x : mpinterval ): mpinterval;    { Logarithm, base 10 }
GLOBAL FUNCTION loga( x, a : mpinterval ) : mpinterval;  { Logarithm, base a }
GLOBAL FUNCTION exp( x : mpinterval ) : mpinterval;   { Exponential function }
GLOBAL FUNCTION exp2( x : mpinterval ) : mpinterval;    { Power fct., base 2 }
GLOBAL FUNCTION exp10( x : mpinterval ) : mpinterval;  { Power fct., base 10 }
GLOBAL FUNCTION power( x, y : mpinterval ) : mpinterval;
GLOBAL FUNCTION arg( x, y : mpinterval ) : mpinterval;
GLOBAL FUNCTION arcsin( x : mpinterval ) : mpinterval;            { Arc sine }
GLOBAL FUNCTION arccos( x : mpinterval ) : mpinterval;          { arc cosine }
GLOBAL FUNCTION arctan( x : mpinterval ) : mpinterval;         { Arc tangent }
GLOBAL FUNCTION arccot( x : mpinterval ) : mpinterval;       { arc cotangent }
GLOBAL FUNCTION arctan2( x, y : mpinterval ) : mpinterval;     { arctan(x/y) }
GLOBAL FUNCTION cos( x : mpinterval ) : mpinterval;                 { Cosine }
GLOBAL FUNCTION sin( x : mpinterval ) : mpinterval;                   { Sine }
GLOBAL FUNCTION tan( x : mpinterval ) : mpinterval;                { Tangent }
GLOBAL FUNCTION cot( x : mpinterval ) : mpinterval;              { Cotangent }
GLOBAL FUNCTION sinh( x : mpinterval ) : mpinterval;       { Hyperbolic sine }
GLOBAL FUNCTION cosh( x : mpinterval ) : mpinterval;     { Hyperbolic cosine }
GLOBAL FUNCTION tanh( x : mpinterval ) : mpinterval;    { Hyperbolic tangent }
GLOBAL FUNCTION coth( x: mpinterval ) : mpinterval;   { Hyperbolic cotangent }
GLOBAL FUNCTION arsinh( x : mpinterval ) : mpinterval;      { Inv. hyp. sine }
GLOBAL FUNCTION arcosh( x : mpinterval ) : mpinterval;    { Inv. hyp. cosine }
GLOBAL FUNCTION artanh( x : mpinterval ) : mpinterval;   { Inv. hyp. tangent }
GLOBAL FUNCTION arcoth( x : mpinterval ) : mpinterval; { Inv. hyp. cotangent }
{============================================================================}
{==========   Module Initalization Part   ===================================}
{============================================================================}
{==============  Last line of module mpi_ari: 2052  =========================}
Number OF rows processed: 2052
```

As may be seen by the preceding listing a large variety of common operators and elementary mathematical functions are realized in this module for multi-precision interval computations.

7.3 How to Use the Multi-Precision Modules

This section describes primarily the allocation and freeing of memory for multi-precision data (see [126]).

The basic data type `mpreal` is dynamically realized as a pointer type. The same is true for the components `inf` and `sup` of the multi-precision interval data type `mpinterval`:

```
global mpinterval = global record inf, sup: mpreal end;
```

In both cases it is in the programmer's responsibility to allocate and free memory used for multi-precision data. The modules `mp_ari` and `mpi_ari` provide the following procedures for memory management:

- `mpinit`: to allocate variables of type `mpreal` or `mpinterval` depending on the actual precision setting,
- `mpfree`: to free the multiprecision variables,
- `mpvlcp`: to generate a temporay copy of function, procedure or operator parameters (`mpvlcp` is an abbreviation for multi-precision value copy) and
- `mptemp`: to mark a multi-precision variable to be temporarily used, that is it will be freed after it is used for the next time.

These procedures can not be used to allocate and free the result variable of a function or operator definition. This is due to a restriction of standard Pascal. Allocating or freeing of result variables using the procedures above would result in a syntax error at compile time. To solve this problem, the modules make use of a programming trick. Since PASCAL-XSC allows overloading of the assignment operator, one can define a special assignment for arguments of type `mpreal` or `mpinterval`, which itself calls the routines listed above. For this purpose, both modules provide an assignment operator with the following right-hand side:

- InitFunc, InitOp: to allocate the result variable, and
- TempFunc, TempOp: to mark the result variable to be temporary (that is the variable will be freed when it will have been used for the next time).

Equipped with the utilities just described, we can give five basic rules for implementing programs using the multi-precision arithmetics provided by the modules `mp_ari` and `mpi_ari`.

R1: **Variables** of type `mpreal` or `mpinterval` must be initialized by a call of `mpinit` and should be freed by a call of `mpfree`:

```
VAR
  x: mpreal;
  y: mpinterval;
BEGIN
  mpinit(x); mpinit(y);
  ...
  mpfree(x); mpfree(y);
END;
```

R2: **Value parameters** of type `mpreal` or `mpinterval` must be saved by a call of `mpvlcp` and must be restored by a call of `mpfree`:

```
PROCEDURE DoSomething(x: mpreal);
BEGIN
  mpvlcp(x);
  ...
  mpfree(x);
END;
```

R3: **Reference parameters** must neither be initialized by `mpinit` nor be freed by `mpfree`.

R4: The **result variable of a function** of type `mpreal` or `mpinterval` must be initialized by an assignment of the value `InitFunc` and should be freed by an assignment of the value `TempFunc`:

```
FUNCTION ComputeSomething(...): mpreal;
BEGIN
  ComputeSomething:= InitFunc;
  ...
  ComputeSomething:= TempFunc;
END;
```

R5: The **result variable of an operator** of type `mpreal` or `mpinterval` must be initialized by an assignment of the value `InitOp` and should be freed by an assignment of the value `TempOp`:

```
OPERATOR + (...) add: mpreal;
BEGIN
  add:= InitOp;
  ...
  add:= TempOp;
END;
```

We now proceed with a variety of examples.

7.4 Examples

In the following subsections we demonstrate the power of the multiple-precision modules mp_ari and mpi_ari.

7.4.1 Logistic Equation

The first example reconsiders the dynamic system already discussed in Paragraph 6.5. A program similar to the program ICHAOS is given. The staggered operations are replaced by multi-precision operations as they are implemented in the modules mp_ari and mpi_ari. In contrast to staggerd operations the precision now is no longer limited by the exponent range of the underlying floating-point system.

The formula realized by the program CHAOS is the mean value form (6.14)

$$\begin{array}{l} X_{n+1} = a \cdot (y_n(1 - X_n) + (1 - 2X_n) \cdot (X_n - y_n)) \\ \quad \text{with} \quad y_n \;\approx\; \mathrm{mid}(X_n) \;=\; \text{midpoint of } X_n \quad . \end{array} \tag{7.2}$$

of the logistic equation (6.13)

$$x_{n+1} = a \cdot x_n \cdot (1 - x_n)\ , \quad n \;\geq\; 0 \tag{7.3}$$

The program performs a multi-precision interval iteration to compute an orbit of the logistic equation with parameter $a = 3.75$, starting point $x_0 = 0.5$. We compute 500 iterations using a precision setting of 10.

```
PROGRAM MPICHAOS;  { Logistic Equation Using Multi-Precision Comp. }
USE i_ari,    { Interval operations                  }
    mpi_ari; { Multi-precision interval arithmetic }

VAR nn    : integer;     { Precision setting }
    x, xm: mpinterval;
    z     : INTERVAL;
    a, x0: REAL;
    n, m  : INTEGER;
BEGIN
   writeln;
   WRITELN('Computation of the logistic equation');
   WRITELN('  x(n+1) = a * x(n) * ( 1-x(n) )');
   WRITELN;
   WRITE('Precision setting (e.g. 10): '); READ(nn);
   setprec(nn);
   mpinit(x); mpinit(xm);           { Initialize multi-prec. variables }
   WRITE('Parameter a (e.g. 3.75): '); READ(a);
   WRITE('Initial value x0 (e.g. 0.5): '); READ(x0);
   WRITE('Number of iterations m (e.g. 500): '); READ(m);
   WRITELN('        Flp. result                       Enclosure');
   x:= x0;               { Initial value for multi-precision iteration }
   FOR n:= 1 TO m DO BEGIN
     x0:= a*x0*(1-x0);                    { Compute real approximation }
     xm:= mid(x);
     x := a*( xm*(1-xm) + (1-2*x)*(x-xm) );     { Multi-prec. enclosure }
     { The line above realizes the mean value form of the logistic eq. }
     z:=x;
     WRITELN(n:4, ': ', x0:14, ' ', z);
   END;
   mpfree(x); mpfree(xm);            { Deallocate multi-prec. variables }
END.
```

The printout shows an enclosure of x_{500}. Please compare the result with the enclosure given in the last line of Table 6.1.

```
Computation of the logistic equation
  x(n+1) = a * x(n) * ( 1-x(n) )

Precision setting (e.g. 10):          10
Parameter a (e.g. 3.75):            3.75
Initial value x0 (e.g. 0.5):         0.5
Number of iterations m (e.g. 500):   500

        Flp. result                          Enclosure
   1:  9.375000E-001 [  9.375000000000000E-001,  9.375000000000000E-001 ]
   2:  2.197266E-001 [  2.197265625000000E-001,  2.197265625000001E-001 ]
   3:  6.429255E-001 [  6.429255008697509E-001,  6.429255008697511E-001 ]
          ...                    ...                       ...
 498:  8.177940E-001 [  5.687786680004961E-001,  5.687786680004963E-001 ]
 499:  5.587763E-001 [  9.197606056052906E-001,  9.197606056052908E-001 ]
 500:  9.245451E-001 [  2.767538774320484E-001,  2.767538774320486E-001 ]
```

A simple modification of the program can be used to find the maximum number $n_{\max}$ of iterations which can be performed with multi-precision interval arithmetic with a specific precision setting. The following table is similar to Table 6.2. The entries in the first column show the precision setting in use. In the second column the corresponding values $n_{\max}$ are listed. The values are given for the mean value form (7.2).

setprec	$n_{\max}$ with (6.14)
10	601
20	1176
30	1754
50	3017
100	6185
150	9243
200	12277

Table 7.1: max. iteration count with multiple-precision arithmetic

Because the orbit of the dynamical system is a subset of the interval $[0, 1]$ we stop the iteration if the enclosure of x_n is no longer a subset of $[0, 1]$.

7.4.2 Multiple-Precision Horner Scheme

We now consider Horner's scheme for the evaluation of the value of a polynomial. We will implement a method to compute the value of a polynomial with at least one ulp accuracy (with respect to the IEEE double data format). We assume that the polynomial coefficients as well as the argument are exactly representable as multi-precision quantities.

Usually, the evaluation of a polynomial $p(x)$ is performed by means of Horner's scheme:

$$p(x) = \sum_{k=0}^{n} a_k x^k = (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0 \tag{7.4}$$

To get sharp bounds for $p(x)$ we will implement a multi-precision interval version. The precision setting local to the implemented function will be automatically increased until an accuracy corresponding to 53 mantissa bits is reached. The multi-precision enclosure is rounded to the smallest floating-point interval u containing this enclosure. Then we check whether `succ(inf(`u`)) >= sup(`u`)`. If this requirement is not satisfied, the precision setting of the multi-precision arithmetic will be increased and a better enclosure will be computed by a repetition of Horner's scheme. If this enclosure does not lead to a floating-point interval u satisfying the inequality from above, the precision setting will be increased once more. This process will be repeated until the bounds of the corresponding ordinary floating-point interval are equal or two adjacent floating-point numbers.

Some Experiments with Fibonacci-Polynomials

To test our routine we will use a so called Fibonacci polynomial. Let f_n denote the n-th Fibonacci number, that is

$$f_0 := 0, \quad f_1 := 1, \tag{7.5}$$

$$f_n := f_{n-1} + f_{n-2}, \quad n \geq 2 \tag{7.6}$$

The first numbers are $0, 1, 1, 2, 3, 5, 8, \ldots$. By induction it can be shown that for any two adjacent Fibonacci numbers f_n, f_{n+1} the relation

$$-f_n^6 - 3f_{n+1}f_n^5 + 5f_{n+1}^3 f_n^3 - 3f_{n+1}^5 f_n + f_{n+1}^6 \in \{-1, \ +1\} \tag{7.7}$$

holds. The value of the expression is $+1$ if n is even and -1 if n is odd.

We now define the Fibonacci polynomial $p_n(x)$ in the following way:

$$p_n(x) := -x^6 - 3f_{n+1}x^5 + 5f_{n+1}^3 x^3 - 3f_{n+1}^5 x + f_{n+1}^6 \tag{7.8}$$

Then relation (7.7) shows that

$$p_n(f_n) = \begin{cases} +1 & \text{if } n \text{ is even} \\ -1 & \text{if } n \text{ is odd} \end{cases} \tag{7.9}$$

The polynomial $p_n(x) = \sum_{m=0}^{6} a_m x^m$ is a polynomial of degree 6. Its coefficients are given by $a_0 := f_{n+1}^6$, $a_1 := -3f_{n+1}^5$, $a_2 := 0$, $a_3 := 5f_{n+1}^3$, $a_4 := 0$, $a_5 := -3f_{n+1}$, $a_6 := -1$.

We will now test our polynomial evaluator using Fibonacci polynomials as input data. In order to handle a point problem we must be sure that the coefficients are computed exactly. This requires an exact computation of the Fibonacci numbers and their respective powers. This is done using the routine `Fibonacci` and the power operator `**` implemented in the module `hornermp`. When creating the polynomial coefficients, the precision setting is increased until all coefficients are representable as multiprecision point intervals. We first list the main program.

```
{----------------------------------------------------------------------}
PROGRAM HornFib;
USE i_ari;     { Real interval arithmetic      }
USE hornermp;  { Multi-precision Horner scheme }
VAR
  k        : integer;          { Loop variable                     }
  n        : integer;          { n-th Fibonacci number is used     }
                               {  as argument of the polynomial    }
  fibo_n   : mpinterval;       { n-th Fibonacci number             }
  fibo_np1 : mpinterval;       { n plus first Fibonacci number     }
  res      : interval;         { Enclosure of pol(fibo_n)          }
  fn, fnp1 : interval;         { Auxiliary variables               }
  pol      : polynom[0..6];    { Polynomial of degree 6            }
  thin     : boolean;          { Indicates whether pol is thin     }

{----------------------------------------------------------------------}
BEGIN
  { Initialization of multi-precision quantities }
  mpinit(fibo_n); mpinit(fibo_np1);
  FOR k:= Lb(pol) TO Ub(pol) DO mpinit( pol[k] );
  { Infinite loop, quit by pressing <Control> <C> }
  REPEAT
    writeln;
    write('The n-th Fibonacci number will be used as argument; n=? ' );
    read(n); writeln;
    setprec(2);
    { Generate polynomial coefficients as exactly representable  }
    { multi-precision numbers (point intervals)                  }
    REPEAT
      setprec(getprec+1);   { Increase precision of mp arithmetic }
      fibo_n  := fibonacci(n);
      fibo_np1:= fibonacci(n+1);
      pol[6]:= -1;
      pol[5]:= -3*fibo_np1;
      pol[4]:=  0;
      pol[3]:=  5*(fibo_np1**3);
      pol[2]:=  0;
      pol[1]:= -3*(fibo_np1**5);
      pol[0]:=  fibo_np1**6;
      thin:= true;   { Check whether all coefficients of pol are thin }
      FOR k:= 0 TO 6 DO BEGIN
        thin:= thin  AND  ( diam(pol[k]) = 0.0 );
      END;
    UNTIL thin;
    { Now the polynomial coefficients are point mp-intervals }
    writeln('Required precision to get point coefficients: ', getprec:0);
    res:= Horner( pol, fibo_n );
    writeln('Enclosure of pol(...):     ', res);
    { Check whether the Fibonacci numbers are representable as    }
    {    IEEE double numbers; if so, print them in a short format }
    writeln('Fibonacci numbers used: ');
    fn  := fibo_n;
    fnp1:= fibo_np1;
    IF diam(fnp1) +> diam(fn) = 0 THEN BEGIN
      writeln('  fibo(',n:3,  '): ', inf(fn):0:1);
      writeln('  fibo(',n+1:3,'): ', inf(fnp1):0:1);
    END
    ELSE BEGIN
      writeln('fibo(n):   ', fibonacci(n));
      writeln('fibo(n+1): ', fibonacci(n+1));
    END;
  UNTIL false; { Break by pressing <Control> <C> }
END.
{----------------------------------------------------------------------}
```

Please notice, that each element of the coefficient array representing the polynomial has to be initialized separately. The data type `polynomial` is defined as a global dynamic type in the module `hornermp`. We now list this module:

```
{---------------------------------------------------------------------}
MODULE hornermp;

USE GLOBAL
  i_ari,   { Real interval arithmetic                        }
  mp_ari,  { Multi-precision real module                     }
  mpi_ari; { Multi-precision interval module                 }

{ New global data type for polynomials with multi-precision }
{   coefficients.                                           }
GLOBAL TYPE polynom = GLOBAL DYNAMIC ARRAY[*] OF mpinterval;

{---------------------------------------------------------------------}
GLOBAL FUNCTION Horner(VAR pol: polynom; VAR pnt: mpinterval): interval;
VAR i, degree, prec, OldPrec, NewPrec: integer;
  r: mpinterval;            { Local multi-precision interval variable. }
  res : interval;
  stop: boolean;
BEGIN
  degree:= UB(pol);
  OldPrec:= getprec;  { Save precision setting.                 }
  mpinit(r);          { Initialize auxiliary mp-variable.       }
  SetPrec(3);         { Start Horner's scheme with precision 3. }
  REPEAT
    writeln('Precision setting in function Horner: ', GetPrec);
    r:= pol[degree];
    FOR i:= degree-1 DOWNTO 0 DO r:= r*pnt + pol[i];
    res:= r;
    writeln('  Intermediate result res: ', res);
    stop:= succ(inf(res)) >= sup(res);  { One ulp accuracy }
    IF NOT stop THEN BEGIN
      NewPrec:= GetPrec+1;  { Increase precision setting }
      SetPrec(NewPrec);
    END;
  UNTIL stop;
  { Now the result is at least of one ulp accuracy with respect to }
  {  the IEEE double format.                                       }
  setprec(OldPrec); { Restore precision setting }
  Horner:= res;
  mpfree(r);    { Deallocate auxiliary variable }
END;

{---------------------------------------------------------------------}
GLOBAL OPERATOR ** (VAR x: mpinterval; n: integer) res: mpinterval;
VAR     { power operator;                   }
        {   only integer values n >= 0 are valid }
  k      : integer;
  pwr, z : mpinterval;
  stop   : boolean;
BEGIN
  mpinit(pwr); mpinit(z);
  pwr:= 1;     { corresponding to x**0 }
  k:= n;
  z:= x;
  stop:= n = 0;
  WHILE NOT stop DO BEGIN
    IF odd(k) THEN pwr:= pwr*z;
    k:= k DIV 2;
    stop:= k = 0;
```

```
    IF NOT stop THEN z:= z*z;
  END;
  res:= InitOp;  { Initialize the mp-result of the operator. }
  res:= pwr;     { Return the value x**n.                    }
  res:= TempOp;  { Set temporary flag of the result.         }
  mpfree(pwr); mpfree(z);   { Deallocate local mp-variables. }
END;

{-----------------------------------------------------------------------}
GLOBAL FUNCTION Fibonacci(n: integer): mpinterval;
VAR                                { Computes the n-th Fibonacci number }
  fib, fib_km1, fib_k: mpinterval;
  k: integer;
BEGIN
  mpinit(fib); mpinit(fib_km1); mpinit(fib_k);
  IF n = 0 THEN
    fib:= 0
  ELSE
    IF n = 1 THEN
      fib:= 1
    ELSE BEGIN
      fib_km1:= 0;
      fib_k  := 1;
      FOR k:= 2 TO n DO BEGIN
        fib    := fib_km1 + fib_k;
        fib_km1:= fib_k;
        fib_k  := fib;
      END;
    END;
  Fibonacci:= InitFunc;  { Initialize the mp-result of the function. }
  Fibonacci:= fib;       { Assign function result.                   }
  Fibonacci:= TempFunc;  { Set temporary flag of the result.         }
  mpfree(fib);           { Deallocation of local multi-precision     }
  mpfree(fib_km1); mpfree(fib_k);               {       quantities. }
END;
{-----------------------------------------------------------------------}
BEGIN
  { Nothing to initialize }
END.
{-----------------------------------------------------------------------}
```

All functions and operators implemented in this module handle multiprecision intervals. In general, the precision setting for the computations is determined by the main program. Only the global function `Horner` manipulates the precision setting for intermediate results by itself. The active precision setting when entering this function is restored before leaving the function. In the body of the function the setting is increased until the enclosure of the correct result is not wider than two units in the last place with respect to the IEEE double format. To check the width of the result, the intermediate multiprecision interval is converted to an ordinary floating point interval. For this interval the accuracy requirement is checked. If the exponent of the multiprecision interval is too large or too small the conversion raises an appropriate error message.

In general, the call of `Horner` will result in an infinite loop if the coefficients of the polynomial are no point intervals. This requirement is not checked when entering this routine. In our application, we force (in the main program, when generating these quantities) the diameters of all coefficients as well as the diameter of the argument to be zero.

A sample output of the program is as follows:

```
The n-th Fibonacci number will be used as argument; n=?   20
Required precision to get point coefficients: 3
Precision setting in function Horner: 3
  Intermediate result res: [ 1.000000000000000E+000, 1.000000000000000E+000 ]
Enclosure of pol(...):     [ 1.000000000000000E+000, 1.000000000000000E+000 ]
Fibonacci numbers used:
  fibo( 20): 6765.0
  fibo( 21): 10946.0


The n-th Fibonacci number will be used as argument; n=?    75
Required precision to get point coefficients: 10
Precision setting in function Horner: 3
  Intermediate result res: [             -2.2E+072,               1.2E+073 ]
Precision setting in function Horner: 4
  Intermediate result res: [              4.5E+062,               3.6E+063 ]
Precision setting in function Horner: 5
  Intermediate result res: [              2.0E+053,               9.3E+053 ]
Precision setting in function Horner: 6
  Intermediate result res: [              2.0E+043,               1.9E+044 ]
Precision setting in function Horner: 7
  Intermediate result res: [              4.1E+033,               4.4E+034 ]
Precision setting in function Horner: 8
  Intermediate result res: [             -5.2E+024,               3.9E+024 ]
Precision setting in function Horner: 9
  Intermediate result res: [             -4.3E+009,               0.0E+000 ]
Precision setting in function Horner: 10
  Intermediate result res: [-1.000000000000000E+000,-1.000000000000000E+000 ]
Enclosure of pol(...):     [-1.000000000000000E+000,-1.000000000000000E+000 ]
Fibonacci numbers used:
  fibo( 75): 2111485077978050.0
  fibo( 76): 3416454622906707.0
```

For the second example (the 75-th Fibonacci number is incorporated) the precision setting local to the function `Horner` is increased 7 times. A final precision setting according to 10 mantissa digits (with respect to base 2^{32}) gives the point interval $[-1,-1]$ as enclosure of the correct polynomial value -1.

7.4.3 Arithmetic-Geometric Mean Iteration

In the next section 7.4.4, the computation of π to high accuracy will be described. As we will see, the computations may be carried out using the so called arithmetic-geometric mean (AGM) iteration. Let g_0 and a_0 be two positive numbers with $0 < g_0 < a_0$. The geometric mean of these numbers is $g_1 := \sqrt{g_0 a_0}$ while their arithmetic mean is $a_1 := \dfrac{g_0 + a_0}{2}$. In general g_j is defined by $g_j := \sqrt{g_{j-1}a_{j-1}}$ and a_j by $a_j := \dfrac{g_{j-1} + a_{j-1}}{2}$, $j = 1, 2, \ldots$. There holds the following monotonicity property

$$g_0 < g_1 < g_2 < \cdots < g_j < \cdots < a_j < \cdots < a_3 < a_2 < a_1 < a_0. \tag{7.10}$$

The sequences $\{g_j\}$ and $\{a_j\}$ converge to their common limit

$$\lim_{j\to\infty} g_j = \lim_{j\to\infty} a_j = AGM(g_0, a_0). \tag{7.11}$$

The rate of convergence is two.

The AGM method is *not* a self-correcting method like the Newton iteration. All iteration steps have to be performed with a full length arithmetic. The following program is a straightforward implementation in order to get verified bounds for the arithmetic-geometric mean of two positive real numbers. Multiple-precision interval arithmetic is used.

```
PROGRAM AGM;                 { Arithmetic-Geometric Mean Evaluation }

USE mp_ari, mpi_ari;       { Multi-precision real and interval module }

FUNCTION AGM(x, y: mpinterval): mpinterval;
VAR
  a, b, bnew: mpinterval;
BEGIN
  mpvlcp(x); mpvlcp(y);  { Value copy for mpinterval value parameter }
  mpinit(a); mpinit(b); mpinit(bnew);
                        { Initialization of local mpinterval numbers }
  a:= x; { Starting values for the AGM iteration }
  b:= y;
  WHILE a >< b DO BEGIN  { Iterate as long as a and b are disjoint }
    bNew:= sqrt(a*b);    { Geometric mean  }
    a:= (a+b)/2;         { Arithmetic mean }
    b:= bNew;
    { In each iteration step the convex hull of a and b is an }
    {   enclosure of AGM(x,y) = AGM(y,x).                      }
  END;
  AGM:= InitFunc;                     { Initialize function result }
  AGM:= a +* b;                           { Convex hull of a and b }
  AGM:= TempFunc;   { Result of function is only temporarily used }
  mpfree(a); mpfree(b); mpfree(bnew); { Free memory for local var }
  mpfree(x); mpfree(y);
END;

VAR
  a, b, res: mpinterval;  { Multi-precision intervals }
  nn, relerr: integer;
BEGIN
  mpinit(a); mpinit(b); mpinit(res); { Initialize mpinterval variables }
  writeln; writeln('*** Arithmetic-Geometric Mean Evaluation ***');
  writeln         ('***    to Arbitrary Verified Accuracy    ***');
  REPEAT
    writeln; write('Number of mantissa digits (base=2**32)  ?   ');
    read(nn); writeln;
    setprec(nn); { Precision setting }
    { An mpinterval is read by its bounds (without brackets) }
    write('a, b  (e.g. 1 1 for a and 2 2 for b): ');
    read(a, b); writeln;
    res:= AGM(a, b);
    IF nn < 7 THEN writeln(res);
    setprec(2);
    relerr:= 1 + expo( (res.sup -> res.inf) /> res.inf );
    writeln('Rel. error of enclosure <= 2**(', relerr, ')' );
  UNTIL false; { Infinite loop; stop with <ctrl> <c> }
END.
```

A sample output of this program is shown below. An enclosure for the arithmetic-geometric mean $AGM(1,2)$ is computed. The input `1 1` denotes the degenerate interval [1,1] (point interval). A lower and an upper bound for the arithmetic-geometric mean as well as an upper bound for the relative error of the enclosure are given.

```
*** Arithmetic-Geometric Mean Evaluation ***
***    to Arbitrary Verified Accuracy    ***

Number of mantissa digits (base=2**32)  ?      3
a, b  (e.g. 1 1 for a and 2 2 for b):  1 1   2 2
 1.45679103104690686905624968 92E+000
 1.45679103104690686932730023 24E+000
Rel. error of enclosure <= 2**(-62)

Number of mantissa digits (base=2**32)  ?      6
a, b  (e.g. 1 1 for a and 2 2 for b):  1 1   2 2
 1.4567910310469068691864323832650819749738639432193226906 01E+000
 1.4567910310469068691864323832650819749738639432234280571 97E+000
Rel. error of enclosure <= 2**(-157)

Number of mantissa digits (base=2**32)  ?    100
a, b  (e.g. 1 1 for a and 2 2 for b):  1 1   2 2
Rel. error of enclosure <= 2**(-3165)
```

For example, using 100 mantissa digits for the multiple-precision interval arithmetic gives an enclosure of $AGM(1,2)$ with 3165 correct bits.

7.4.4 Computation of π using Brent's Method

Algorithms for the computation of trigonometric functions in general make use of π or some multiple of π. Especially for multiple-precision implementations of such routines guaranteed bounds of these constants with sufficient accuracy are needed. The computation of guaranteed bounds for elliptic integrals (see [195]) is also based on the availability of guaranteed bounds for π with sufficient accuracy.

The following method is due to Brent [72]. Its rate of convergence is two. As with the AGM method all operations has to be performed with full length. The method is not a self correcting one.

Let $g_0 := \dfrac{\sqrt{2}}{2}$ and $a_0 := 1$. Define

$$t_n := \frac{1}{2} - \sum_{k=0}^{n} 2^{k-1} c_k^2$$

where

$$c_k := \sqrt{a_k^2 - g_k^2} = \frac{c_{k-1}^2}{4a_k} = a_{k-1} - a_k \ , \quad k = 0, 1, \ldots$$

and g_n and a_n are computed by the AGM iteration $AGM(g_0, a_0)$. Then

$$\frac{a_{n+1}^2}{t_n} < \pi < \frac{a_n^2}{t_n} \ , \quad n = 0, 1, \ldots \ .$$

leads to a sequence of enclosures of π. The rate of convergence is two. In each step of the iteration the number of correct mantissa digits is approximately doubled. In the following PASCAL-XSC program the calculations are done using a multiple-precision interval arithmetic.

```
PROGRAM BrentPi;
USE mp_ari, mpi_ari; { modules for multiple-precision arithmetic }
VAR
  a, b, x, y, t, np5 : mpinterval;
  pi_lb, pi_ub       : mpreal;
  nn, iter, err, errold    : integer;
BEGIN
  mpinit(a); mpinit(b); mpinit(x); mpinit(y); mpinit(t);
  mpinit(np5); mpinit(pi_lb); mpinit(pi_ub);
  write('Mantissa digits (base=2**32)? '); read(nn); writeln;
  setprec(nn);
  { nn mantissa digits with respect to base 2**32 are used }
  np5:= 0.5;
  a:= 1.0; b:= sqrt(np5); t:= 0.25; x:= 1.0;
  err:= 0;
  iter:= 0;
  REPEAT
    iter:= iter + 1; { number of actual iteration step }
    errold:= err;
    y:= a;
    a:= (a + b)/2; { arithmetic mean }
    b:= sqrt(y*b); { geometric mean  }
    y:= a - y;
    t:= t - x*y*y; { ti             }
    x:= x + x;
    pi_lb:= inf((a+b)*(a+b)/(4*t));
    pi_ub:= sup(a*a/t);
    { It is pi_lb < pi < pi_ub in each iteration step (see Brent) }
    err:= 1 + expo(pi_ub -> pi_lb);
    { upwardly directed rounded subtraction to get upper bound of }
    { exponent of the difference of upper and lower bound for pi  }
    writeln('Step ', iter:2, ':     Errorbound < 2**(', err, ')');
  UNTIL err > 2*(errold-1);     { no longer quadratic convergence }
  mpfree(np5); mpfree(pi_lb); mpfree(pi_ub);
  mpfree(a); mpfree(b); mpfree(x); mpfree(y); mpfree(t);
END.
```

The program produces the following output:

```
Mantissa digits (base=2**32)?
Step  1:     Errorbound < 2**(-4)
Step  2:     Errorbound < 2**(-13)
Step  3:     Errorbound < 2**(-31)
Step  4:     Errorbound < 2**(-67)
Step  5:     Errorbound < 2**(-140)
Step  6:     Errorbound < 2**(-285)
Step  7:     Errorbound < 2**(-575)
Step  8:     Errorbound < 2**(-1155)
Step  9:     Errorbound < 2**(-2315)
Step 10:     Errorbound < 2**(-4636)
Step 11:     Errorbound < 2**(-4763)
```

In each step the difference of the upper and lower bounds of the enclosure of π is printed. The absolute value of the exponent (to base 2) is a lower bound of the correct number of bits reached so far. The 10-th iterate is an approximation of π

with 4636 correct bits. For the last step the number of correct bits is not doubled. This is due to the limited precision (in this example 150 mantissa digits with respect to the base 2^{32}).

Other methods for the computation of π may be found in [63], [64], [202] and [294].

7.4.5 Newton's Method for two Equations in two Unknowns

In this section we will discuss Newton's method, the classical example for a self correcting method. Under favorable circumstances the rate of convergence is two.

Newton's method applied to the system of nonlinear equations

$$h : \mathbb{R}^n \to \mathbb{R}^n, \quad h(z) = \begin{pmatrix} h_1(z_1, z_2, \ldots, z_n) \\ h_2(z_1, z_2, \ldots, z_n) \\ \vdots \\ h_n(z_1, z_2, \ldots, z_n) \end{pmatrix} \stackrel{!}{=} 0 \tag{7.12}$$

is an iteration designed to find a vector $\hat{z} \in \mathbb{R}^n$ such that

$$h(\hat{z}) = 0$$

simultaneously. The iteration formula (Newton's method) is given by

$$z^{k+1} := z^k - (h'(z^k))^{-1} \cdot h(z^k), \quad k = 0, 1, \ldots \tag{7.13}$$

with a starting vector z^0 which is sufficiently close to the root $\hat{z}$. The Jacobian matrix $h'(z)$ is defined by

$$h'(z) = \begin{pmatrix} \dfrac{\partial h_1}{\partial z_1} & \cdots & \dfrac{\partial h_1}{\partial z_n} \\ \vdots & \cdots & \vdots \\ \dfrac{\partial h_n}{\partial z_1} & \cdots & \dfrac{\partial h_n}{\partial z_n} \end{pmatrix} . \tag{7.14}$$

In case of two equations in two unknowns

$$h(z) = h(z_1, z_2) = \begin{pmatrix} h_1(z_1, z_2) \\ h_2(z_1, z_2) \end{pmatrix} =: \begin{pmatrix} f(x, y) \\ g(x, y) \end{pmatrix}$$

the inverse R of the Jacobian matrix can be shown to be (if it exists)

$$(h'(x, y))^{-1} = \frac{1}{f_x g_y - g_x f_y} \begin{pmatrix} \dfrac{\partial g}{\partial y} & \dfrac{-\partial f}{\partial y} \\ \dfrac{-\partial g}{\partial x} & \dfrac{\partial f}{\partial x} \end{pmatrix} =: R \tag{7.15}$$

evaluated at (x, y). This leads to a reformulation of the iteration formula in a componentwise form:

$$\left.\begin{aligned} x^{k+1} &:= x^k - \frac{f\,g_y \;-\; g\,f_y}{d} \\ y^{k+1} &:= y^k - \frac{g\,f_x \;-\; f\,g_x}{d} \end{aligned}\right\}\; k = 0, 1, 2 \ldots \tag{7.16}$$

with d being

$$d \;:=\; f_x\,g_y \;-\; g_x\,f_y\,. \tag{7.17}$$

All function evaluations as well as the evaluations of the partial derivatives $f_x = \dfrac{\partial f}{\partial x}$, $f_y = \dfrac{\partial f}{\partial y}$, $g_x = \dfrac{\partial g}{\partial x}$ and $g_y = \dfrac{\partial g}{\partial y}$ have to be done at the previous approximate (x^k, y^k).

7.4.5.1 Verification Step

The verification step is based on Brouwer's fixed point theorem. If for an approximation $\tilde{z}$ of the root $\hat{z}$ of $h(z)$ and an interval vector $0 \in Z \in IR$ the condition

$$-R \cdot h(\tilde{z}) + \{I - R \cdot h'(\tilde{z} + Z)\} \cdot Z \subset Z \tag{7.18}$$

holds, then there exists exactly one root $\hat{z} \in \tilde{z} + Z$ with $h(\hat{z}) = 0$ [285]. The real matrix $R = (r_{ij})$ is an approximation for the Jacobian (7.14) of h, and I denotes the identity matrix of corresponding order. All operations are interval operations. For two equations of two variables the condition can be reformulated in a componentwise manner:

If for an approximation $\tilde{z} = (\tilde{x},\ \tilde{y})$ of the root $\hat{z} = (\hat{x},\ \hat{y})$ of $h(z)$ and an interval vector $0 \in Z = (X,\ Y) \in IR$ the condition

$$\left.\begin{aligned} -r_{11}f - r_{12}g + (1 - r_{11}f_x - r_{12}g_x)X - (r_{11}f_y + r_{12}g_y)Y &\;\subset\; X \\ -r_{21}f - r_{22}g + (1 - r_{21}f_y - r_{22}g_y)Y - (r_{21}f_x + r_{22}g_x)X &\;\subset\; Y \end{aligned}\right\} \tag{7.19}$$

holds, then there exists exactly one root $(\hat{x}, \hat{y}) \in \ (\tilde{x} + X,\ \tilde{y} + Y)$ with $h(\hat{x},\ \hat{y}) = 0$. Here, f and g have to be evaluated at $(\tilde{x}, \tilde{y})$ and the partial derivatives at $(\tilde{x} + X,\ \tilde{y} + Y)$. Again, all operations are interval operations.

7.4.5.2 PASCAL-XSC Listing using matrix/vector notation

We first give a simple implementation of the method described above using the predefined matrix/vector data types and matrix/vector operations for ordinary floating-point formats as they are supplied by the PASCAL-XSC modules `mv_ari` and `mvi_ari`. The computation of the inverse of an approximation to the Jacobian is done using predefined routines of the standard modules `lss` and `ilss`. The implementation is based on formulas (7.13) and (7.18). Such a simple matrix/vector implementation is up to now not available for multi-precision operations. Nevertheless, matrix/vector modules for multi-precision data types can be implemented.

```
PROGRAM newt_d2m;
{--------------------------------------------------------}
{                          n        n                     }
{    Root of a function h: R   ---> R  will be enclosed    }
{           by an n-dimensional interval vector.           }
{                                                          }
{--------------------------------------------------------}

USE problem; { Specifications of the nonlinear system }
             {   i.e. of the function h(z).            }

USE lss, ilss;  { Computation of the inverse  of a matrix }

{ Comfortable interface for the computation of an inverse }
FUNCTION inverse(VAR A: rmatrix):
               rmatrix[lb(A,1)..ub(A,1), lb(A,2)..ub(A,2)];
VAR T: imatrix[lb(A,1)..ub(A,1), lb(A,2)..ub(A,2)];
    err: integer;
BEGIN
  inv(A, T, err); { procedure inv is defined in module lss }
  IF err <> 0 THEN writeln('***** E R R O R  in inv !!');
  inverse:= mid(T);
END;
{--------------------------------------------------------}

FUNCTION inverse(VAR A: imatrix):
               rmatrix[lb(A,1)..ub(A,1), lb(A,2)..ub(A,2)];
VAR T: imatrix[lb(A,1)..ub(A,1), lb(A,2)..ub(A,2)];
    err: integer;
BEGIN
  inv(A, T, err);
  IF err <> 0 THEN writeln('***** E R R O R  in inv !!');
  inverse:= mid(T);
END;
{--------------------------------------------------------}

VAR
  { h_dim is the number of components of the considered    }
  { function h. It is defined as a global constant.        }
  it, max_it, i: integer;
  x            : rvector[1..h_dim];   { real iteration     }
  z, y, y_old  : ivector[1..h_dim];   { interval iteration }
  R            : rmatrix[1..h_dim, 1..h_dim];
BEGIN
  write('Starting value for the root?   ');
  FOR i:= 1 TO h_dim DO read(x[i]);
  writeln;
  it:= 0;
  max_it:= 7;
  REPEAT
    it:= it + 1;
    x:= x - inverse( J(x) ) * h(x); { Newton step }
    writeln('it: ', it:1, '  h(actual approximate):');
    writeln(h(x));
    readln;
  UNTIL it = max_it;

  R:= inverse(J(x));  { Approximate inverse of the Jacobian }
  z:= -R*h(intval(x));
  y:= z;
  it:= 0;
  REPEAT
    it:= it + 1;
    y_old:= blow(y, 1.0E-315) +* null(y);
```

```
    y:= z + ( id(r) - R*J(intval(x) + y_old) )*y_old;
    writeln('Interval iteration step: ', it);
    readln;
  UNTIL (y IN y_old) OR (it = max_it);
  IF y IN y_old THEN writeln('Verified enclosure: ', x + y);
END.
```

7.4.5.3 PASCAL-XSC Listing (Componentwise Form)

The PASCAL-XSC program given now is a straightforward implementation of the formulas (7.16), (7.17). In each step of the iteration the user is prompted to change the precision of the computations. For each step the new estimates of the roots are printed as well as the values of the component functions f(x,y) and g(x,y) evaluated at these new estimates. Under favourable circumstances the iteration gives a sequence of iterates which converges to the root quadratically. In this case the number of mantissa digits used may be doubled for succeding iteration steps.

```
PROGRAM Newt_Mp;
{---------------------------------------------------------}
{                            2        2                    }
{    Root of a function h: R    ---> R  will be enclosed   }
{           by a 2-dimensional interval vector.            }
{                                                          }
{---------------------------------------------------------}

USE
  mp_ari,     { Multiple-precision real arithmetic          }
  mpi_ari,    { Multiple-precision interval arithmetic      }
  mp_out,     { Special printing of mult.-prec. intervals   }
  prob;       { Specifies the system of nonliear equations  }

VAR
  it, i, nn, max_expo, max_it         : integer;
  x, y, x0, y0, d                     : mpreal;
  xpu, ypv, u, v, u_new, v_new,z1, z2 : mpinterval;
  r11, r12, r21, r22, h               : mpinterval;
  incl: boolean;

BEGIN
  mpinit(u); mpinit(u_new); mpinit(z1);
  mpinit(v); mpinit(v_new); mpinit(z2);
  mpinit(xpu); mpinit(ypv);
  mpinit(r11); mpinit(r12); mpinit(r21); mpinit(r22);
  mpinit(x); mpinit(y); mpinit(x0); mpinit(y0);
  mpinit(d);
  mpinit(h);

  write('Approximation for the root ?  '); read(x,y); writeln;

  { Real iteration }
  it:= 0;
  max_it:= 10; { maximum number of Newton steps }
  REPEAT
    write('Precision ?   '); read(nn);
    setprec(nn);
    writeln('  ', nn:2);
    it:= it + 1;

    d:= fx(x,y)*gy(x,y) - gx(x,y)*fy(x,y);
```

```
    x:= x -(f(x,y)*gy(x,y) - g(x,y)*fy(x,y)) / d;

    d:= fx(x,y)*gy(x,y) - gx(x,y)*fy(x,y);
    y:= y -(g(x,y)*fx(x,y) - f(x,y)*gx(x,y)) / d;

    max_expo:= max( expo(f(x,y)), expo(g(x,y)) ) + 1;
    writeln('Step: ', it:1,'    |f|, |g| < 2**(', max_expo:1, ')');
  UNTIL it = max_it;

  { Interval iteration for verification }
  d:= fx(x,y)*gy(x,y) - gx(x,y)*fy(x,y);
  r11:=  gy(x,y)/d;
  r12:= -fy(x,y)/d;
  r21:= -gx(x,y)/d;
  r22:=  fx(x,y)/d; { Approximate Inverse of the Jacobian }
  u:= x;
  v:= y;
  z1:= -r11*f(u,v) - r12*g(u,v);
  z2:= -r21*f(u,v) - r22*g(u,v);
  u_new:= z1;
  v_new:= z2;

  it:= 0;
  max_it:= 10; { Maximum number of interval iteration steps }
  REPEAT
    writeln;
    it:= it + 1;

    u:= blow(u_new);
    v:= blow(v_new);

    xpu:= _mpinterval(x)+*(x+u); { Convex hull of x and x+u }
    ypv:= _mpinterval(y)+*(y+v);

    u_new:= z1 + (1.0 - r11*fx(xpu, ypv) - r12*gx(xpu, ypv))*u
                - (r11*fy(xpu, ypv) + r12*gy(xpu, ypv))*v;
    v_new:= z2 + (1.0 - r21*fy(xpu, ypv) - r22*gy(xpu, ypv))*v
                - (r21*fx(xpu, ypv) + r22*gx(xpu, ypv))*u;

    { Check inclusion property }
    incl:= (u_new IN u) AND (v_new IN v);
    IF incl THEN BEGIN
      writeln('!!!!! Enclosure of the solution:');
      writeln('x-coordinate:');
      writeln(_mpinterval(x) + u_new:3);
      writeln('y-coordinate:');
      writeln(_mpinterval(y) + v_new:3);
    END
    ELSE BEGIN
      writeln('Iteration step: ', it:1, '   No inclusion so far!');
    END;
    writeln; write('Press <Enter> to proceed!   '); readln;
  UNTIL incl OR (it = max_it);

  mpfree(h);
  mpfree(d);
  mpfree(x); mpfree(y); mpfree(x0); mpfree(y0);
  mpfree(r11); mpfree(r12); mpfree(r21); mpfree(r22);
  mpfree(xpu); mpfree(ypv);
  mpfree(v); mpfree(v_new); mpfree(z2);
  mpfree(u); mpfree(u_new); mpfree(z1);
END.
```

The specification of the system of nonlinear equations is done in a separate module called `prob`:

```
MODULE prob;   { main program is: newt_mp.p }
USE
  mp_ari,    { multiple-precision real arithmetic }
  mpi_ari;   { multiple-precision interval arithmetic }
GLOBAL CONST
  c1 = 0.125; { parameters of                     }
  c2 = 0.125; {  the considered nonlinear system }
{ These constants are global constants for the component       }
{ global functions f(x,y) and g(x,y) of                         }
{ the system of two nonlinear equations in two unknowns        }
{                                                               }
{                   (  f(x, y)  )     ( exp(-x+y) - c1 )   !    }
{   h(z) = h(x,y) = (           ) :=  (                )   = 0  }
{                   (  g(x, y)  )     ( exp(-x-y) - c2 )        }
{                                                               }
{ The exact solution (root) of the given system is             }
{                                                               }
{                     ( -0.5 ln(c1*c2) )                        }
{                 z:= (                )                        }
{                     (  0.5 ln(c1/c2) )                        }
{                                                               }

{ Functions and partial derivatives for real arguments: }

GLOBAL FUNCTION f(VAR x, y: mpreal): mpreal;
BEGIN
  f:= true;
  f:= exp(-x+y) - _mpreal(c1);
  f:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION g(VAR x, y: mpreal): mpreal;
BEGIN
  g:= true;
  g:= exp(-x-y) - _mpreal(c2);
  g:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION fx(VAR x, y: mpreal): mpreal;
BEGIN
  fx:= true;
  fx:= -exp(-x+y);
  fx:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION fy(VAR x, y: mpreal): mpreal;
BEGIN
  fy:= true;
  fy:= exp(-x+y);
  fy:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION gx(VAR x, y: mpreal): mpreal;
BEGIN
  gx:= true;
  gx:= -exp(-x-y);
  gx:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION gy(VAR x, y: mpreal): mpreal;
```

```
BEGIN
  gy:= true;
  gy:= -exp(-x-y);
  gy:= false;
END;
{-----------------------------------------------------------}

{ Functions and partial derivatives for interval arguments: }
{  (overloading of global functions)                         }

GLOBAL FUNCTION f(VAR x, y: mpinterval): mpinterval;
BEGIN
  f:= true;
  f:= exp(-x+y) - _mpinterval(c1);
  f:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION g(VAR x, y: mpinterval): mpinterval;
BEGIN
  g:= true;
  g:= exp(-x-y) - _mpinterval(c2);
  g:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION fx(VAR x, y: mpinterval): mpinterval;
BEGIN
  fx:= true;
  fx:= -exp(-x+y);
  fx:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION fy(VAR x, y: mpinterval): mpinterval;
BEGIN
  fy:= true;
  fy:= exp(-x+y);
  fy:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION gx(VAR x, y: mpinterval): mpinterval;
BEGIN
  gx:= true;
  gx:= -exp(-x-y);
  gx:= false;
END;
{-----------------------------------------------------------}
GLOBAL FUNCTION gy(VAR x, y: mpinterval): mpinterval;
BEGIN
  gy:= true;
  gy:= -exp(-x-y);
  gy:= false;
END;
{-----------------------------------------------------------}
END. {of the module prob }
```

7.4.5.4 Numerical Example

As an example the following system of nonlinear equations is considered:

$$h(z) = \begin{pmatrix} f(x,y) \\ g(x,y) \end{pmatrix} := \begin{pmatrix} \exp(x+y) & - & c_1 \\ \exp(-x-y) & - & c_2 \end{pmatrix} \overset{!}{=} \begin{pmatrix} 0 \\ 0 \end{pmatrix} . \tag{7.20}$$

The parameters c_1 and c_2 are real quantities. The exact solution vector $\hat{z} \in I\!R^2$ of this system is given by

$$\hat{z} = \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} := \begin{pmatrix} -0.5 \cdot \ln(c_1 \cdot c_2) \\ -0.5 \cdot \ln\left(\dfrac{c_1}{c_2}\right) \end{pmatrix}. \tag{7.21}$$

The program based upon matrix/vector operations given in Section 7.4.5.2 leads to the following sample output. The exact solution of the problem of the first run (here we use the parameters $c_1 := 1.125, c_2 = 0.125$) is $(-0.5\ln(9/64),\ -0.5\ln(9))^T = (\ln(8/3),\ -\ln(3))^T$. The program produces the following output:

```
Starting value for the root?       1   0
it: 1  h(actual approximate):
 6.509814806340075E-002
 1.755797187372329E+000
it: 2  h(actual approximate):
 9.975809295851401E-003
 4.410929666446322E-001
it: 3  h(actual approximate):
 3.597300542156345E-004
 5.667405156220262E-002
it: 4  h(actual approximate):
 5.156441406362067E-007
 1.337597591980932E-003
it: 5  h(actual approximate):
 1.063510390864053E-012
 7.939268307222136E-007
it: 6  h(actual approximate):
 2.775557561562891E-017
 2.799982468104645E-013
it: 7  h(actual approximate):
-2.775557561562891E-017
 0.000000000000000E+000
Interval iteration step: 1
Interval iteration step: 2
Verified enclosure:
[  9.808292530117260E-001,  9.808292530117266E-001]
[ -1.098612288668111E+000, -1.098612288668109E+000]
```

In the second run the parameters c_1 and c_2 are both set to the value 0.125. So the correct solution is given by $(-0.5\ln(1/64),\ 0)^T = (\ln(8),\ 0)^T$. The printout of the program is as follows:

```
Starting value for the root?  1  1       c1=c2=0.125
it: 1  h(actual approximate):
 2.918620196785084E-001
 3.847852820374753E-004
it: 2  h(actual approximate):
 8.197845189639522E-002
 5.898178638830132E-007
it: 3  h(actual approximate):
 1.428779943748035E-002
 1.391525783489556E-012
it: 4  h(actual approximate):
 7.083768377087463E-004
-2.775557561562891E-017
it: 5  h(actual approximate):
```

```
 1.992136587219973E-006
 2.775557561562891E-017
it: 6  h(actual approximate):
 1.587410758396857E-011
 2.775557561562891E-017
it: 7  h(actual approximate):
 2.775557561562891E-017
 2.775557561562891E-017
Interval iteration step: 1
Interval iteration step: 2
Interval iteration step: 3
Verified enclosure:
[  2.079441541679835E+000,  2.079441541679837E+000]
[              -4.9E-016,                 1.3E-016]
```

Now the program using multiple-precision interval arithmetic described in paragraph 7.4.5.3 is executed to solve the same problem (parameters $c_1 = c_2 := 0.125$). For the final iteration steps of the real iteration the required precision of the calculations has been almost doubled in each step. This is adequate to the quadratic rate of convergence. The notation

```
  Step: 1    |f|, |g| < 2**(-2)
```

means that after one step of the interation the magnitudes of both component functions f and g evaluated at the actual approximate are smaller than 2^{-2} (for the exact solution of the nonlinear system the component functions are 0 exactly).

```
Approximation for the root ?       1   1
Precision ?      3
  Step: 1    |f|, |g| < 2**(-2)
Precision ?      3
Step: 2    |f|, |g| < 2**(-4)
Precision ?      3
Step: 3    |f|, |g| < 2**(-8)
Precision ?      3
Step: 4    |f|, |g| < 2**(-15)
Precision ?      3
Step: 5    |f|, |g| < 2**(-29)
Precision ?      3
Step: 6    |f|, |g| < 2**(-56)
Precision ?      5
Step: 7    |f|, |g| < 2**(-111)
Precision ?      9
Step: 8    |f|, |g| < 2**(-220)
Precision ?     17
Step: 9    |f|, |g| < 2**(-439)
Precision ?     33
Step: 10    |f|, |g| < 2**(-877)

Iteration step: 1   No inclusion so far!
  Press <Enter> to proceed!

!!!!! Enclosure of the solution:
x-coordinate:
  [ 2.0794415416798359282516963643745297042265
        0040308076576236204002848018086590908414
          6817589980989256062626004443061712057201
            0565607072743916710980122549052278857921
              8271248511430557092111587167502041337005
                0346093493865717361425518044724796195466
                  5742024185776488513590212859790999279623
```

```
                    1271747344485613122129115649632,50745 E+000  ]

  y-coordinate:
    [-5.563E-309   , 5.181E-318  ]
```

The final enclosure is accurate to more then 300 decimal places. The correctness of the enclosure is verified by the algorithm itself (Brouwer's fixed-point theorem in combination with interval computations). Using the multi-precision interval logarithm supplied by the module `mpi_ari` we can verify that the x-coordinate displayed above is an enclosure of the first component $\ln(8)$ of the exact solution vector. The exact value of the second component is zero and indeed, the y-coordinate printed above is an enclosure of zero.

7.4.6 Arcsine Function

In this section we consider an implementation of the inverse sine function for multi-precision real arguments. The resulting multi-precision interval will contain the exact function value. The accuracy of the enclosure is related to the actual precision setting for the multi-precision point argument. We do not consider rounding errors by an a priory error estimation. Instead we use multi-precision interval arithmetic to find an enclosure of the truncated Taylor series. Only a bound for the relative approximation error caused by the truncation of Taylor's series has to be derived.

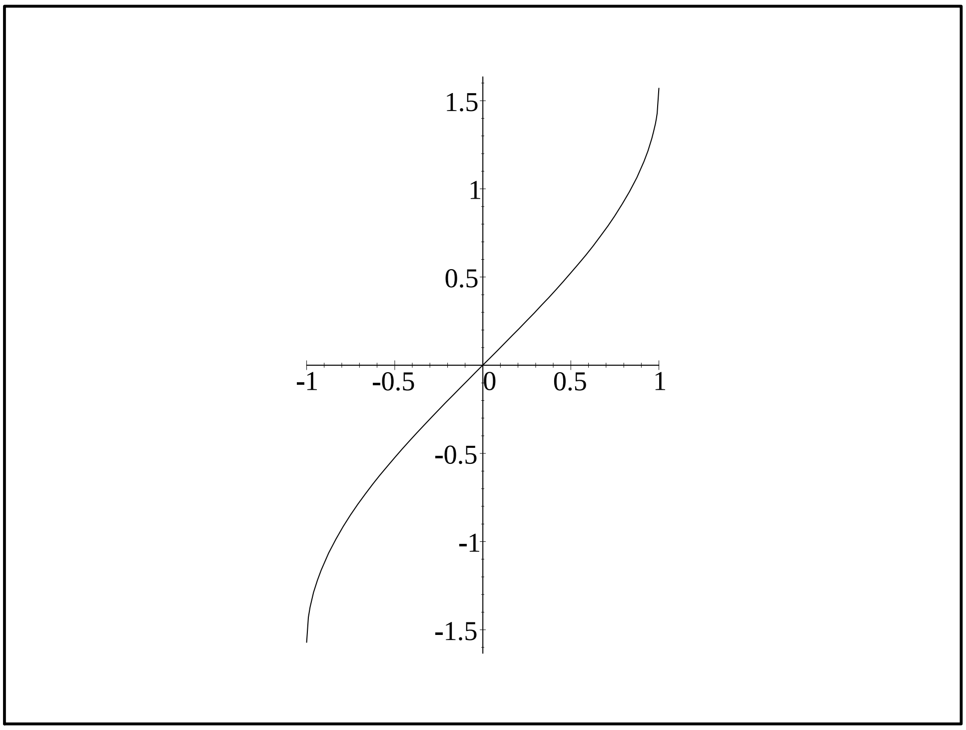


Figure 7.1: The inverse sine function $\arcsin(x)$

Mathematical Background The inverse sine function can be defined by

$$\arcsin x = \sum_{k=0}^{\infty} q_k \cdot x^{2k+1} =: s_\infty(x) \tag{7.22}$$

for $|x| \le 1$. The first coefficients q_k are given by

$$\begin{aligned} q_0 &:= 1, \\ q_1 &:= \frac{1}{2 \cdot 3}, \\ q_2 &:= \frac{1 \cdot 3}{2 \cdot 4 \cdot 5}, \\ q_3 &:= \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6 \cdot 7}, \\ q_4 &:= \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9}, \end{aligned}$$

and, in general, by

$$q_k := \frac{(2k)!}{(2^k k!)^2 \cdot (2k+1)} = \frac{1}{2k+1} \prod_{j=1}^{k} \frac{2j-1}{2j} \,. \tag{7.23}$$

They are all positive and the sequence $\{q_k\}_{k=0}^{\infty}$ is monotone decreasing (it holds $q_{k+1}/q_k < 1,\ k = 0, 1, \ldots$). Therefore, the truncated series

$$s_n = s_n(x) := \sum_{k=0}^{n} q_k \cdot x^{2k+1} \tag{7.24}$$

is monotone increasing in x for values x in the range $0 \le x \le 1$. Thus, for $[\underline{x}, \overline{x}] \subseteq [0, 1]$ it holds

$$\begin{aligned} \bigwedge_{x \in [\underline{x}, \overline{x}]} s_n(x) &= \sum_{k=0}^{n} q_k \cdot x^{2k+1} \\ &\in \left[\sum_{k=0}^{n} q_k \cdot \underline{x}^{2k+1}, \sum_{k=0}^{n} q_k \cdot \overline{x}^{2k+1} \right] = [s_n(\underline{x}), s_n(\overline{x})]. \end{aligned} \tag{7.25}$$

Later on, the coefficients q_{n+1} we will be computed by

$$q_{n+1} = q_n \cdot \frac{(2n+1) \cdot (2n+1)}{(2n+2) \cdot (2n+3)} \,. \tag{7.26}$$

The relative approximation error can be estimated using

$$|\arcsin x| \ge |x| \,. \tag{7.27}$$

We find

$$
\begin{aligned}
\varepsilon_n(x) &:= \left| \frac{\arcsin(x) - x \cdot \sum_{k=0}^{n} q_k \cdot x^{2k}}{\arcsin(x)} \right| \\
&= \left| \frac{x}{\arcsin x} \right| \sum_{k=n+1}^{\infty} q_k \cdot x^{2k} \\
&\le \sum_{k=n+1}^{\infty} q_k \cdot x^{2k} \\
&= q_{n+1} \cdot x^{2n+2} \sum_{k=n+1}^{\infty} \frac{q_k}{q_{n+1}} \cdot \frac{x^{2k}}{x^{2n+2}} \\
&= q_{n+1} \cdot x^{2n+2} \left\{ 1 + \underbrace{\frac{q_{n+2}}{q_{n+1}}}_{\in(0,1)} \cdot x^2 + \underbrace{\frac{q_{n+3}}{q_{n+1}}}_{\in(0,1)} \cdot x^4 + \dots \right\} \\
&\le q_{n+1} \cdot x^{2n+2} \sum_{k=0}^{\infty} x^2 = q_{n+1} \cdot x^{2n+2} \cdot \frac{1}{1 - x^2}
\end{aligned}
\tag{7.28}
$$

This bound is valid for $x \in (-1, 1)$. Here the geometric series with common ratio $x^2 < 1$ has been applied. For $[\underline{x}, \overline{x}] \subseteq [0, 1/2]$ it holds

$$
\bigwedge_{x \in [\underline{x}, \overline{x}]} \varepsilon_n(x) \overset{(7.28)}{\le} q_{n+1} \cdot \frac{4}{3} \cdot x^{2n+2} \le \frac{4}{3} \cdot q_{n+1} \cdot \overline{x}^{2n+2} \tag{7.29}
$$

The right hand side of (7.29) is an overestimation of the maximum relative truncation error. The special values $x \in \{-1,\ 0,\ 1\}$ will be treated in a correct way as may be seen by the source code of the program module shown in Paragraph 7.4.6.3. In the case $x = 0$ the point interval [0,0] will be returned. In the remaining cases enclosures for $\pm\frac{\pi}{2}$ depending on the sign of x.

7.4.6.1 Argument Reduction

The argument reduction process uses the relations

$$
\arcsin(x) = -\arcsin(-x) \tag{7.30}
$$

as well as

$$
\arcsin(x) = \frac{\pi}{2} - 2 \cdot \arcsin \sqrt{\frac{1-x}{2}} \tag{7.31}
$$

The first relation is used if $x < 0 \quad (\Rightarrow -x > 0)$ and the second one is used if $|x| > 1/2 \quad (\Rightarrow \sqrt{(1-|x|)/2} \in [0, 1/2])$. Thus the final reduced argument t lies in the basic range $0 \le t \le 1/2$. In this range (7.29) can be used.

7.4.6.2 Algorithmic Description

We first discuss an algorithm to compute a lower bound of $\arcsin(x)$.

Algorithm 7.1: Asin (x) {function}

$\varepsilon := 10^{-k}$, k depends on the actual precision setting when entering this function.
neg:= $x < 0$ {neg=true if the argument is smaller then zero}
if neg **then**
  $[r] := -x$
else
  $[r] := x$
reduc := $[r] > 0.5$
if reduc **then** $[r] := \sqrt{\dfrac{1-[r]}{2}}$ {argument reduction}

$n := 0$
repeat {until truncation error is small enough}
  $[s] := s_n([r]) = \sum_{k=0}^{n} q_k[r]^{2k+1}$ {truncated Taylor series}
  RelErr:= $\triangle(\frac{4}{3} \cdot q_{n+1} \cdot \overline{r}^{2n+2})$ { see formula (7.29)}
  {$\triangle$ denotes the upwardly directed rounding}
  $n := n + 1$

until RelErr$< \varepsilon$

$[s] := [s] \cdot [1, 1 + \text{RelErr}]$ {take truncation error into account}

if reduc **then**
  $[s] := [\frac{\pi}{2}] - 2 \cdot [s]$ {apply (7.31)}
if neg **then**
  $[res] := -[s]$ {apply (7.30)}
else
  $[res] := [s]$

return Asin $:= [res]$

7.4.6.3 Program Code

The following module `mpasin.p` implements Algorithm 7.1.

```
MODULE mpasin;
```

```
{ This module implements the inverse sine functuion called asin(x)   }
{ x being a multi-precision real number. The result will be a multi- }
{ precision interval which contains the exact function value.        }


USE i_ari;            { 'normal' interval arithmetic }
USE mp_ari, mpi_ari;  { multi-precision arithmetics  }

GLOBAL FUNCTION asin(x: mpreal): mpinterval;
{ The name of the intrinsic multi-precision arcsine function is }
{ arcsine. Therefore we use the different name asin.            }
VAR
  RelErr: real;
  r, rr, rpot, s, e, q_n: mpinterval;
  c, err, ir, irpot, iq_n: interval;
  n: integer;
  neg, reduc: boolean;

BEGIN
  mpvlcp(x);
  mpinit(r); mpinit(rr); mpinit(rpot); mpinit(s);
  mpinit(e); mpinit(q_n);

  { Check whether x is a valid argument in the reange [-1, 1] }
  IF (x<-1) OR (1<x) THEN BEGIN
    writeln('*** Invalid argument in asin(x) in module mpasin.p!');
    n:= n DIV 0; { force program termination by a divide by zero }
  END;

  RelErr:= exp(-getprec*9*ln(10.0));
  { A precision setting of getprec corresponds to roughly 8*getprec }
  { decimal digits. So 10**(-9*getprec) is used as error bound for  }
  { the relative approximation error.                               }

  setprec(getprec+1);  { one guard digit with respect to base 2**32 }
  neg:= x < 0;
  IF neg THEN  { r = abs(x) }
    r:= -x
  ELSE
    r:= x;

  reduc:= (r.sup > 0.5); { true if an argument reduction is applied }
  IF reduc THEN
    r:= sqrt( 0.5*( 1.0 - r ) );

  { Notice: x in the set (-1, 0, 1)   ==> r = 0 }

  { The basic interval for the reduced argument r is [0, 0.5] }
  rr:= r*r;
  rpot:= r;
  ir:= r;
  c:= 4.0 / intval(3.0);   { 'normal' interval }

  s:=rpot;      { s=q_0 * r }
  n:= 1;
  q_n:= 1.0 / _mpinterval(6.0);  { = q_1 = 1/(2*3) }

  REPEAT
    rpot:= rpot * rr;
    s:= s + q_n * rpot;
    q_n:= q_n * ( (2*n+1)*(2*n+1) ) / _mpinterval ( (2*n+2)*(2*n+3) );
    n:= n+1;
    { It holds:                        }
    {                   (2*n+1)*(2*n+1)    }
```

```
    {     q       =  q  *  ---------------    }
    {      n+1        n    (2*n+2)*(2*n+3)    }

    iq_n := q_n;      { conversion to a 'normal' interval }
    irpot:= rpot;     { conversion to a 'normal' interval }

    Err:= c * iq_n * irpot * ir; { 'normal' interval arithmetic }
  UNTIL Err.sup < RelErr;

  e.inf:= 1;
  e.sup:= e.inf +> Err.sup;   { e:= [1, 1+Err.sup] }
  s:= s*e;
  IF reduc THEN { perform result adaptation }
     s:= ( arctan( _mpinterval(1) ) - s )*2;
  { arctan(1) = pi/4; the argument 1 is handled as a special value.     }
  { For a value of getprec not to large, a stored constant is returned. }

  setprec(getprec-1);  { restore initial precision setting }
  asin:= InitFunc;  { initialize function result }
  IF neg THEN
    asin:= intval(-s.sup, -s.inf)  { actual argument was negative }
  ELSE
    asin:= intval(s.inf, s.sup);
  asin:= TempFunc;  { set temporary flag }

  mpfree(q_n); mpfree(e);   { deallocate mp-variables }
  mpfree(r); mpfree(rr); mpfree(rpot); mpfree(s);
  mpfree(x);
END;

BEGIN  { body of the module mpasin.p }
  { nothing to inizialize }
END.
```

A corresponding driver program is given now:

```
PROGRAM asintst;
{***    Driver program for the module mpasin.p   ***}
USE mp_ari, mpi_ari;   { multi-precision arithmetic }
USE mpasin;            { new module                 }

VAR i: integer;
    x: mpreal;         { multi-precision point argument  }
BEGIN
  mpinit(x);  { initialize mp-variable }
  FOR i:= 1 TO 5 DO BEGIN
    setprec(2+i); { increase actual precision setting }
    write('Argument for arcsin:  ');
    read(x);
    writeln;
    writeln('Argument: ');
    writeln(x);
    { The number of decimal digits shown in the output depends on }
    {   the actual precision setting. }
    writeln('Actual precision setting: ', getprec:0);
    { The function getprec gives the number of mantissa digits }
    {   with respect to base 2**32.                            }
    writeln('asin(x)  (lower bound, upper bound): ');
    writeln(asin(x));
   END;
   mpfree(x);  { deallocate mp-variable }
END.
```

The program will produce the following output:

```
Argument for arcsin:   0.125
Argument:
 1.2500000000000000000000000000E-001
Actual precision setting: 3
asin(x)  (lower bound, upper bound):
 1.2532783116806539687456698632E-001
 1.2532783116806539687456698636E-001
Argument for arcsin:   0.125
Argument:
 1.25000000000000000000000000000000000000E-001
Actual precision setting: 4
asin(x)  (lower bound, upper bound):
 1.2532783116806539687456698635708471802 6E-001
 1.25327831168065396874566986357084718054E-001
Argument for arcsin:  -0.125
Argument:  -
-1.250000000000000000000000000000000000000000000000E-001
Actual precision setting: 5
asin(x)  (lower bound, upper bound):
-1.253278311680653968745669863570847180481477268426E-001
-1.253278311680653968745669863570847180481477268248E-001
Argument for arcsin:   0.75
Argument:
 7.500000000000000000000000000000000000000000000000000000000E-001
Actual precision setting: 6
asin(x)  (lower bound, upper bound):
 8.480620789814810080529443389984180800733662132631126428262E-001
 8.480620789814810080529443389984180800733662132631126430351E-001
Argument for arcsin:   0.96875
Argument:
 9.687500000000000000000000000000000000000000000000000000000000000000E-001
Actual precision setting: 7
asin(x)  (lower bound, upper bound):
 1.3201406644587658254821877189255820060022892460102081600194377181024E+000
 1.3201406644587658254821877189255820060022892460102081600194377181025E+000
```

Here we can immediately see that the accuracy of the enclosure depends on the precision setting done by `setprec()` in the driver program `asintst.p` .

7.4.7 Creation of Constants for Interval Functions

In this section we will give some hints how to compute constants (for example coefficients of polynomials, reduction constants for function evaluations, values like $\ln(2), \ln(10)$, π, and so on) to maximum accuracy with respect to the IEEE double floating-point format or even with an accuracy beyond the accuracy associated with that data format. In the latter case a sequence of floating-point values will be generated, the sum of which is very close to the exact value of the desired constant, i. e. we will compute a kind of staggered representation (see also chapter 5.3.5).

Let us first compute $\ln(10) = 2.302585\ldots$ to 53 bit accuracy. Our goal is to find $t = \Box(\ln(10))$. As usual, the symbol $\Box$ denotes the operation round to the nearest IEEE double number. Of course we could take the value of t (the correct bit sequence) from a table. Due to the hidden bit technique of the IEEE data format and exponent considerations cumbersome bitmanipulations would be necessary. Moreover, constants which are not so common can in general not be found in such a table.

Due to the availability of a lot of mathematical functions for multi-precision real and interval arguments in PASCAL-XSC it is often very easy to find the floating-point representation of a real value with maximum accuracy. We only have to round the bounds of a multi-precision enclosure of the desired value to ordinary floating-point numbers. This can be done using the overloaded assignment operator `:=` . This operator performs a conversion (round-to-nearest!) from a multi-precision value to a floating-point number. If the conversion of the lower bound of the enclosure is equal to the conversion of the upper bound, it follows that the converted lower bound is the round to nearest value of the constant.

The following listing shows a PASCAL-XSC program for the computation of the correctly rounded value of ln(10):

```
PROGRAM ln10;
USE i_ari,   { interval arithmetic }
    mp_ari,  { real multi-precision arithmetic }
    mpi_ari, { interval multi-precision arithmetic }
    x_real;  { hexadecimal printing of real numbers (IEEE convention) }

VAR
  Ln10Long: mpinterval; { Multi-prec. enclosure of ln(10) }
  Ln10: real;           { ln(10) round to nearest         }
  a, b: real;           { Auxiliary real values           }
BEGIN
  setprec(2);
  mpinit(Ln10Long);
  REPEAT
    setprec( getprec+1 );  { Increase precision setting                }
    Ln10Long:= ln( _mpinterval(10) );
    a:= inf(Ln10Long);  { Lower bound round to the nearest IEEE number }
    b:= sup(Ln10Long);  { Upper bound round to the nearest IEEE number }
  UNTIL  a = b;
  Ln10:= a;                       { a = b = round to nearest( ln(10) ) }
  writeln('Ln(10) round to nearest: ');
  writeln( Ln10, '   IEEE notation: ', Ln10:'X');
  writeln('Precision setting used: ', getprec:1);
  mpfree(Ln10Long);
END.
```

The output of this program is as follows:

```
Ln(10) round to nearest:
 2.302585092994046E+000   IEEE notation: 40026BB1BBB55516
Precision setting used: 3
```

The next program demonstrates how to approximate a transcendental number (here π) to high accuracy using a sum of floating point values:

```
PROGRAM LongConst;
USE i_ari,   { interval arithmetic }
    mp_ari,  { real multi-precision arithmetic }
    mpi_ari, { interval multi-precision arithmetic }
    x_real;  { hexadecimal printing of real numbers (IEEE convention) }

CONST kMax = 15;   { number of summands, upper bound of loop variable }

VAR PiDown,  { high prec. lower bound of Pi }
    PiUp:    { high prec. upper bound of Pi }
      ARRAY[1..kMax] OF real;
```

```
    x, Pi: mpinterval;  { auxiliary multi-prec. numbers }
    y: interval;        { used for rounding purposes    }
    k: integer;
BEGIN
  setprec( 2*kMax+2 );
  mpinit(x);   { Initialize multi-prec. variables }
  mpinit(Pi);
  x:= 1;              { arctan(1) = Pi/4 }
  Pi:= 4*arctan(x);  { Enclosure of Pi  }
  x:= Pi;             { Initialize x with an enclosure of Pi }

  writeln(' k          PiDown (dec)        PiDown (hex)      PiUp (hex)');
  writeln('==============================================================');
  FOR k:= 1 TO kMax DO BEGIN
     y:= x;  { round multi-prec. variable x to an ordinary interval }
     PiDown[k]:= inf( y );  { take lower bound of y as new summand  }
     IF k < kMax THEN
        PiUp[k]  := inf( y )
     ELSE { Only the last summands of PiDown and PiUp are different }
        PiUp[k] := sup( y );
     writeln(k:2, '   ', PiDown[k], '  ',
                               PiDown[k]:'X', '  ', PiUp[k]:'X' );
     x:= x - PiDown[k];
     IF 0 IN x THEN writeln('No more significant digits!');
  END;
  mpfree(x);    { Deallocate multi-prec variables }
  mpfree(Pi);
END.
```

The sum of the floating-point values in column PiDown is a high precision lower bound whereas the sum of the quantities in column PiUp is a high precision upper bound for the constant π:

k	PiDown (dec)	PiDown (hex)	PiUp (hex)
1	3.141592653589793E+000	400921FB54442D18	400921FB54442D18
2	1.224646799147353E-016	3CA1A62633145C06	3CA1A62633145C06
3	2.165713347843828E-032	395C1CD129024E08	395C1CD129024E08
4	1.479700953653540E-048	36014CF98E804177	36014CF98E804177
5	2.525569933693951E-064	32BA98EC4E6C8945	32BA98EC4E6C8945
6	5.288548833776613E-081	2F4410F31C6809BB	2F4410F31C6809BB
7	8.162387396922563E-097	2BFBE5466CF34E90	2BFBE5466CF34E90
8	8.072565166210643E-113	28A8D98158536F92	28A8D98158536F92
9	1.120904125742092E-128	255F14374FE1356D	255F14374FE1356D
10	5.471661597409286E-145	21FB54709179216D	21FB54709179216D
11	2.600518365932964E-161	1E976625E7EC6F44	1E976625E7EC6F44
12	3.025811221032718E-177	1B4885D34C6FDAD6	1B4885D34C6FDAD6
13	4.108789252345320E-194	17C7FEB96DE80D6F	17C7FEB96DE80D6F
14	5.224700962386170E-210	147B7B8E1AFED6A2	147B7B8E1AFED6A2
15	2.741499313753246E-226	1119FA5AE9F24117	1119FA5AE9F24118

The last example in this section deals with the representation of floating point quantities by quotients of exactly representable decimal numbers. The numerator will be an integer value and the denominator a suitable power of 2. Such representations can be used to specify polynomial coefficients, reduction constants, best possible approximations of transcendental constants and so on by decimal strings. No hexadecimal input is necessary. The compiler transforms automatically and without any rounding error the numerator as well as the denominator in the IEEE floating-point format. The final division is nothing else but a shift. It also will be done error free. The program, as it is shown here, does not resort to multi-precision calculations. However, multi-precision calculations will be used in general to find a good approximation for the exact constant.

```
PROGRAM quot;
{ Computation of p and q with p/q = round downwards(ln10) }

USE x_real,    { hexadecimal printing of real numbers }
    i_ari;     { interval arithmetic                  }

VAR c: real;   { Constant to be expressed as p/q }
    p: real;   { numerator                       }
    q: real;   { denominator (power of two)      }
    shift: integer; { Exponent of denominator    }
BEGIN
  c:= inf( ln(intval(10)) ); { Constant to be converted }
  { The constant c usually comes from a multi-prec. computation }
  write('Constant c: ', c, ' (dec)  ', c:'X', ' (IEEE)');
  shift:= 53 - expo(c);    { Reals have 53 mantissa bits }
  { Scaling of c by 2**shift results in an integer value }
  { 0.5 <= c < 1 ==> expo(c) = 0  }
  writeln( '  Exponent of c: ', expo(c) );
  writeln( '  p = c * 2**', shift:2 );
  writeln( '  q =     2**', shift:2 );
  p:= c*power(2, shift); { transform const. c to integer }
  q:=   power(2, shift); { denominator = 2**(52-expo(c)) }
  writeln( 'Numerator p   (down): ',   p:50:15:-1);
  writeln( '              (up  ): ',   p:50:15:+1);
  writeln( 'Denominator q (down): ',   q:50:15:-1);
  writeln( '              (up  ): ',   q:50:15:+1);
  { Check  whether p/q is exactly equal to c in all rounding modes }
  IF c - intval(p)/q <> 0 THEN
    writeln('*** E R R O R ***')
  ELSE BEGIN
    writeln('It holds: ');
    writeln('  Floating-point number c = p/q without rounding error!');
  END;
END.
```

The output of the program shows the representation of the best possible lower bound (IEEE double format) of ln(10):

```
Constant c:  2.302585092994045E+000 (dec)  40026BB1BBB55515 (IEEE)  Expo-
nent of c: 2
  p = c * 2**51
  q =     2**51
Numerator p   (down):                    5184960683398421.000000000000000
              (up  ):                    5184960683398421.000000000000000
Denominator q (down):                    2251799813685248.000000000000000
              (up  ):                    2251799813685248.000000000000000
It holds:
  Floating-point number c = p/q without rounding error!
```

7.5 Concluding Remarks

Due to the operator concept of PASCAL-XSC and the overloading of functions (generic functions) programs using the modules for multiple-precision real and multiple-recision interval arithmetic are easy to develop easy to read and easy to debug. At the present state, variables of the multiple-precision data types have to be initialized (function `mpinit( )`). Also the result of a multiple-precision function or operator has to be initialized and the temporary flag has to be set 'by hand'. Nevertheless, we hope that the numerous examples given above show how easy and straightforward programs for multi-precision computations can be developed using the modules `mp_ari` and `mpi_ari`.

A great advantage of programs written in PASCAL-XSC is the portability of these programs. In the meantime PASCAL-XSC is available for a great variety of computers. Going to a PASCAL-XSC system on another computer no changes have to be made at the source code of the programs. Also the numerical results are bitwise the same. This statement holds true for multi-precision computations in PASCAL-XSC.

7.6 Exercises

Exercise 7.1 The quantity

$$q_1 := e^{\pi\sqrt{163/9}}$$

starts with the decimal expansion

$$q_1 = 640320.00000000\ldots\ .$$

Try to show by computation whether q_1 is an integer.

The same question arises for

$$q_2 := e^{\pi\sqrt{163}}\ .$$

What is the correct value of q_2 up to 50 decimal places? In Brent [73] a Fortran 77 program is printed which computes the quantities above using the multiprecision package described in the same paper. Compare your PASCAL-XSC source code with the one given by Brent.

Exercise 7.2 The argument reduction for the trigonometric functions is usually done by a subtraction of the form

$$x - k \cdot \frac{\pi}{2} \tag{7.32}$$

Here the integer value $k \in Z$ is to be chosen in such a way that the difference (7.32) will be a quantity between $-\pi/4$ and $\pi/4$. Compute for the real value

$$x = x_{\text{bad}} := 6381956970095103 \cdot 2^{797} \tag{7.33}$$

the appropriate $k = k_x$. How many decimal digits has k_x? Compute for the values x_{bad} and $k = k_x$ an enclosure of the difference (7.32). Compute an enclosure of $\cos(x_{\text{bad}})$. What is the best possible enclosure of $\cos(x_{\text{bad}})$ with IEEE double numbers as bounds? Verify that x_{bad} is exactly representable in the IEEE double floating-point format. How many bits cancel due to the process of argument reduction? To which accuracy $\pi/2$ has to be known to get an accurately reduced argument to say 53 bits?

Remark: The value x_{bad} given above is an IEEE double number which comes closest to the set

$$M := \{\ k \cdot \frac{\pi}{2} \mid k \in Z,\ k \neq 0\ \}. \tag{7.34}$$

I. e. if we denote the set of all IEEE double numbers by S it holds (cf. [258])

$$\mathrm{dist}(S\backslash\{0\},\ M) = \mathrm{dist}(\ x_{\text{bad}},\ M)\ . \tag{7.35}$$

Exercise 7.3 Show by induction relation (7.7) on page 214 concerning Fibonacci numbers.

7.7 References and Further Readings

There exists a lot of literature about multiprecision arithmetics. As a main reference see Knuth [186]. One of the most frequently used and most popular package is Brents package [73, 75]. Newer articles are [28, 29, 126, 138, 202, 203, 304]. Some of the described packages are available via anonymous ftp.

Chapter 8

(Pseudo) Random Number Generators

In this paragraph we present some random number generators. PASCAL-XSC intrinsically does not provide any random number generator. Especially when testing implementations of algorithms for mathematical functions such generators are a very useful tool. But of course they are also useful in many other applications.

8.1 Remarks on Linear Congruential Generators

We only will provide pseudo random number generators which are based on a recurrence of the form

$$r_{j+1} = a \cdot r_j + c \,(\text{mod}\, m) \tag{8.1}$$

Such gererators are called linear congruential generators. They generate a sequence of integers $r_1, r_2, r_3, \ldots$, each between 0 and $m-1$. Here the integer m is called the modulus and a and c are positiv integers called the multiplier and the increment respectively. The recurrence (8.1) will eventually repeat itself, with a period that is obviously no greater than m. If m, a, and c are properly chosen, then the period will be of maximal length. In that case, all possible integers between 0 and $m-1$ occur at some point, so any initial 'seed' is as good as any other.
The congruential method has the advantage of being very fast, requiring only a few operations per call hence its almost universal use. It has the disadvantage that it is not free of sequential correlation on successive calls. If k random numbers at a time are used to plot points in k dimensional space (with each coordinate between 0 and 1), then the points will not tend to 'fill' up the k-dimensional space, but rather will lie on $(k-1)$-dimensional 'planes'. There will be at most about $m^{\frac{1}{k}}$ such planes. If the constants m, a and c are not very carefully chosen, they will be many fewer than that. Figures 8.1, 8.2, 8.3 shows the result using the (bad) generator

$$r_{j+1} = 24298 r_j + 99991 \quad (\text{mod } 199017) \tag{8.2}$$

to generate points in $\mathbb{R}^3$. Figure 8.1 gives the impression of rather randomly generated points. But if we change the orientation appropriately we see large regions which do not contain any point generated by our random number generator. All points lie on only 4 planes (see Figure 8.2 and Figure 8.3).

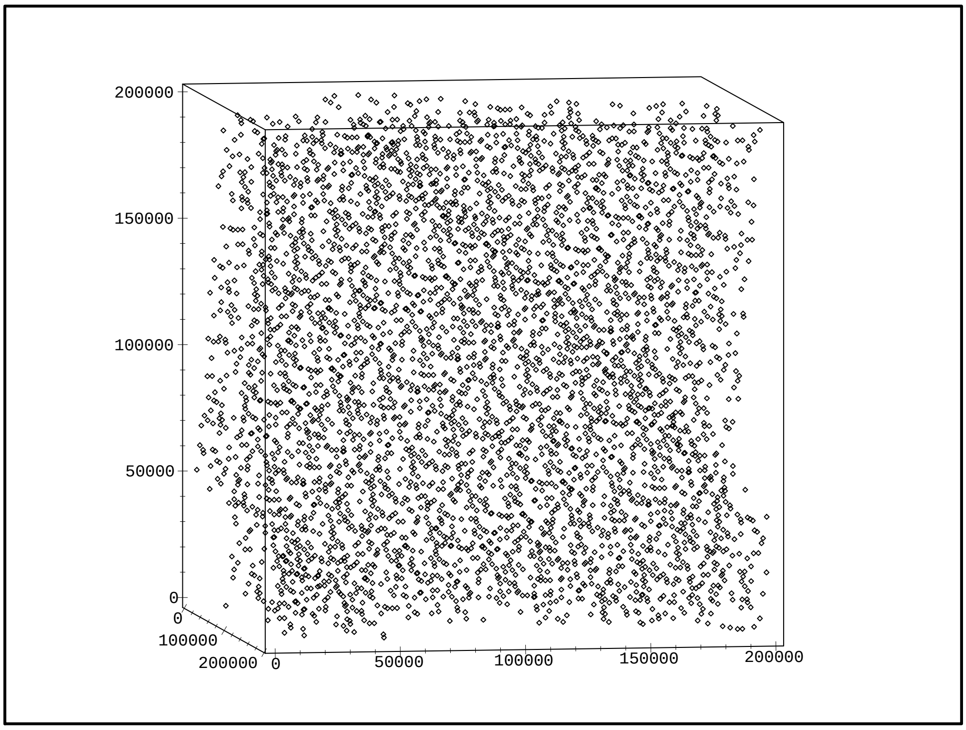


Figure 8.1: Random points generated by the recurrence (8.2)

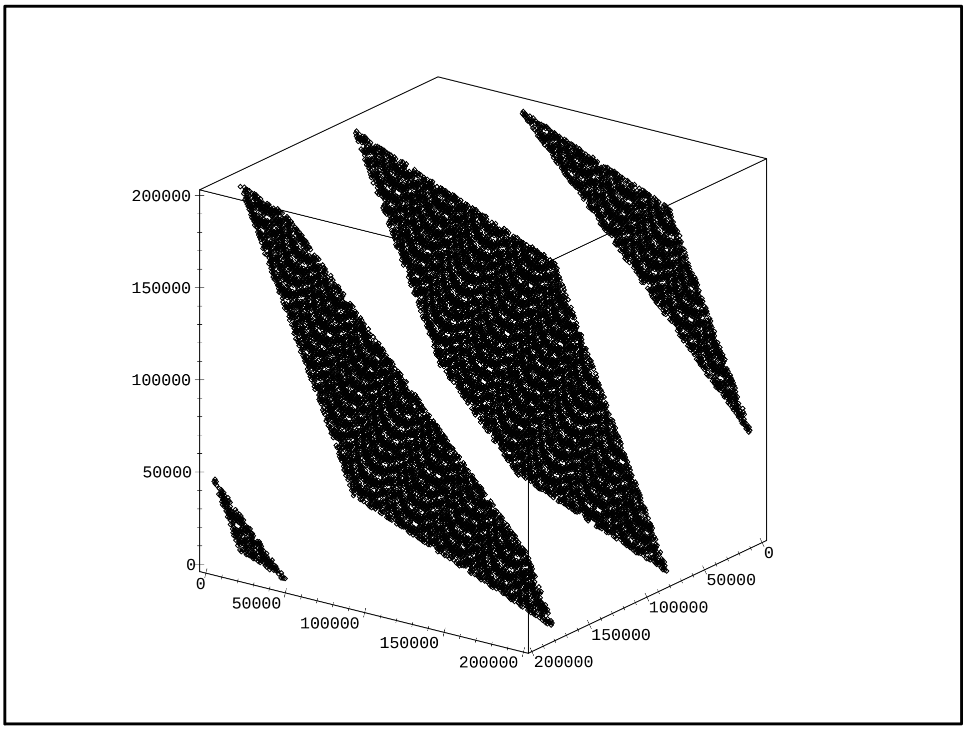


Figure 8.2: Random points generated by recurrence (8.2) lie on only four planes

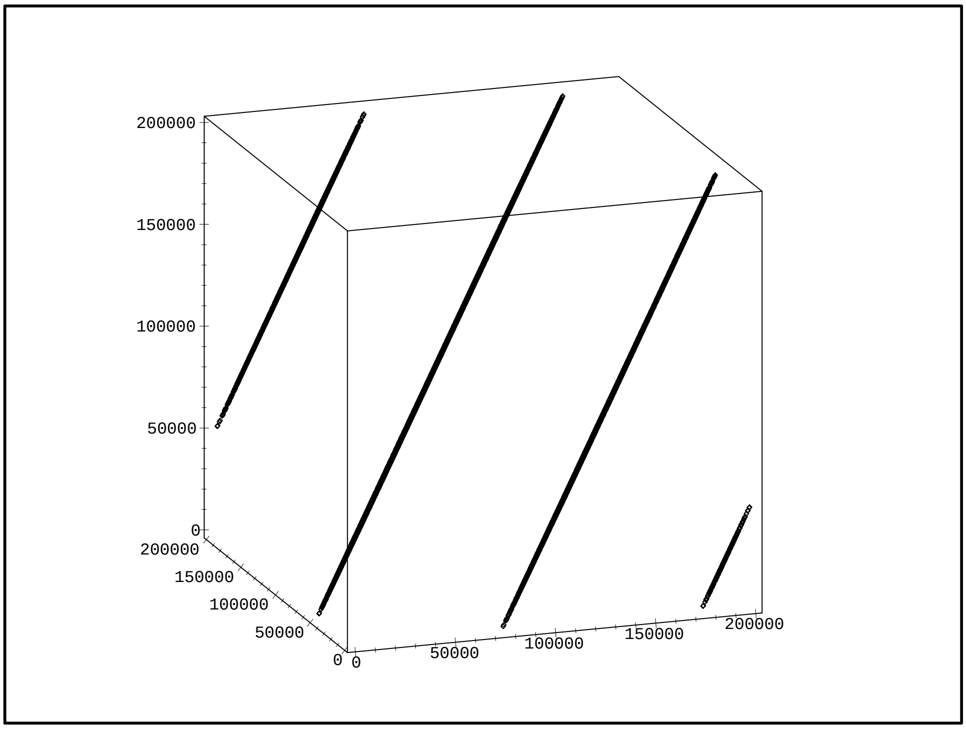


Figure 8.3: Random points generated by the recurrence (8.2) (one additional sight)

8.2 Algorithms

We first give the algorithm for the 'Minimal Standard' generator as it was introduced by Park and Miller [264] and used in [265]. This generator has a vanishing increment $c = 0$. It is given by

$$r_{j+1} = 7^5 \cdot r_j \pmod{2^{31} - 1}, \quad \text{i.e. ,} \tag{8.3}$$

$$m := 2147483647, \; a := 16807, \; c := 0. \tag{8.4}$$

Of course, $r_0 = 0$ must never be allowed as initial seed. For any other initial seed the period of (8.3) is $2^{31} - 2 \approx 2 \cdot 10^9$. To avoid products (e.g. $a \cdot (m-1)$) larger than the maximum value of a 32-bit integer, we use Schrage' s algorithm [298] for multiplying to 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits:
If in the approximate factorization of the multiplier m,

$$m = a \cdot q + b \quad (q := \mathrm{trunc}(\frac{m}{a}), \;\; r := m \bmod a, \text{the values } a \text{ and } m \text{ are given})$$

b is smaller than q and $0 < z < m - 1$, it follows that all intermediate results are representable, i.e. it holds

$$a \cdot (z \bmod q), \;\; b \cdot \mathrm{trunc}(\frac{z}{q}) \in \{0, \ldots, m-1\},$$

and

$$a \cdot z \bmod m = \begin{cases} a \cdot (z \bmod q) - b \cdot \mathrm{trunc}\dfrac{z}{q} & \text{if this value is} > 0\ , \\[2ex] a \cdot (z \bmod q) - b \cdot \mathrm{trunc}\dfrac{z}{q} + m & \text{otherwise.} \end{cases}$$

For the Minimal Standard generator the values $q := 127773$ and $b := 2836$ are used.

Algorithm 8.1: **rand0** {function, no argument}

{Fixed values: $a = 16807, m = 2147483647, w = \dfrac{1}{m}, q = 127773, b = 2836$ }
{ rk (initialized by seed = 123456789) will be changed at each call! }

$j := rk$ div q
$rk := a \cdot (rk - j \cdot q) - b \cdot j$
if $rk < 0$ **then** $rk := rk + m$

return rand0:= $rk \cdot w$

The algorithm given here does not allow an initial seed being supplied by the user. In the PASCAL-XSC program code an additional procedure `seed` is provided. This global procedure can be used to initialize the random number generator. Algorithm rand0 is satisfactory for the majority of applications. Some (in most cases minor) disadvantages (see Press [265], page 279) will be removed by the generators implemented in the modules `rand1` and `rand2`. For the generators we will not give an algorithmic description. We only will give their PASCAL-XSC code in the next section. They are adaptations of the programs given in [265] to PASCAL-XSC. For our routines the naming of variables given there is adopted.

8.3 Program Code

The first module called `rand0` implements the Minimal Standard generator due to Park and Miller:

```
MODULE rand0;
{---------------------------------------------------------------}
{ "Minimal" random number generator of Park and Miller. It returns }
{ a uniform random deviate between 0.0 and 1.0 (exclusive of the   }
{ endpoint values).                                                }
{---------------------------------------------------------------}

VAR
 idum: integer;
 IA, IM, IQ, IR, MASK: integer;
 AM: real;

GLOBAL PROCEDURE Seed(seed: integer);
BEGIN
```

```
  idum:= abs(seed);
END;

GLOBAL FUNCTION Rand: real;
VAR k: integer;
BEGIN
  k:= idum DIV IQ;
  idum:= IA*(idum-k*IQ) - IR*k;
  IF idum < 0 THEN idum:= idum+IM;
  Rand:= AM*idum;
END;

BEGIN
{----- Initialization part ----------------------------------------}
  idum:= 123456789;
  IA:=16807;
  IM:=2147483647;
  AM:=1.0/IM;
  IQ:=127773;
  IR:=2836;
  MASK:=123459876;
END.
{----------------------  End of module Rand0  ---------------------}
```

The module rand1 uses the Minimal Standard generator for its random value, but it shuffles the output to remove low-order serial correlation by an algorithm due to Bays and Durham.

```
MODULE rand1;
{------------------------------------------------------------------}
{ "Minimal" random number generator with Bays-Durham shuffle       }
{------------------------------------------------------------------}
CONST NTAB=32;

FUNCTION max(a, b: integer): integer;
BEGIN IF a >= b THEN max:= a ELSE max:= b END;

FUNCTION min(a, b: real): real;
BEGIN IF a<= b THEN min:= a ELSE min:= b END;

VAR iy, idum: integer;
    iv      : ARRAY [1..NTAB] OF integer;
    iyrand1 : integer;
    ivrand1 : ARRAY [1..NTAB] OF integer;

VAR
  IA, IM, IQ, IR, NDIV: integer;
  AM, EPS, RNMX       : real;

GLOBAL PROCEDURE Seed(seed: integer);
{------------------------------------------------------------------}
{ Set the seed for the "Minimal" random number generator          }
{------------------------------------------------------------------}
BEGIN
  idum:= abs(seed);
  IF idum = 0 THEN idum:= 123;          { avoid 0 as initial value }
END;

GLOBAL FUNCTION Rand: real;
{------------------------------------------------------------------}
{ "Minimal" random number generator of Park and Miller with Bays-  }
{ Durham shuffle and added saveguards. Returns a uniform random    }
{ deviate between 0.0 and 1.0 (exclusive of the endpoint values).  }
```

```
{ The notation is according to W.H. Press et al.,                  }
{     "Numerical Recipes in C", second edition, page 280.          }
{------------------------------------------------------------------}
VAR j, k: integer;
BEGIN
  IF iyrand1=0 THEN BEGIN
    idum:= max(-idum,1);
    FOR j:= NTAB+8 DOWNTO 1 DO BEGIN
      k:= idum DIV IQ;
      idum:= IA*(idum-k*IQ) - IR*k;
      IF idum < 0 THEN idum:= idum+IM;
      IF j<NTAB THEN ivrand1[j]:= idum;
    END;
  END;
  k:= idum DIV IQ;                  { Compute idum without overflows }
  idum:= IA*(idum-k*IQ)-IR*k;       {    by Schrage's method.        }
  IF idum < 0 THEN idum:= idum + IM;
  j:= 1 + iyrand1 DIV NDIV;
  iyrand1:= ivrand1[j];
  ivrand1[j]:= idum;
  rand:= min(AM*iyrand1, RNMX);     { Convert idum to a flp-result }
END;

VAR k: integer;
BEGIN
{------------------------- Initialization part -------------------}
  IA:= 16807; IM:= 2147483647;
  AM:= 1.0/IM;
  IQ:= 127773; IR:= 2836;
  NDIV:= 1+(IM-1) DIV NTAB;
  RNMX:= pred(1.0);  { Largest flp-number less than 1 };
  iy:= 0;
  FOR k:= 1 TO NTAB DO ivrand1[k]:= 0;
  idum:= 123456789;
  iyrand1:= 0;
END.
{------------------------- End of module Rand1  ------------------}
```

If very long random sequences are needed the random number generator implemented in the module `rand2` should be used. Here we realize an algorithm due to L'Ecuyer [225] who has given a good way of combining two different sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods. As in the previous module `rand1` an additional shuffle is performed.

```
MODULE Rand2;
{----------------------------------------------------------------}
{ Long period (>2*1E18) random number generator of L'Ecuyer      }
{----------------------------------------------------------------}

CONST NTAB=32;

VAR
  iy, idum, idum2: integer;
  iv: ARRAY [1..NTAB] OF integer;
  AM, RNMX: real;
  IM1, IM2, IMM1, IA1, IA2, IQ1, IQ2, IR1, IR2, NDIV: integer;
  start: boolean;

FUNCTION max(a, b: integer): integer;
BEGIN IF a >= b THEN max:= a ELSE max:= b END;
```

```
FUNCTION min(a, b: real): real;
BEGIN IF a<= b THEN min:= a ELSE min:= b END;

GLOBAL FUNCTION Rand: real;
{-----------------------------------------------------------------}
{ Long period (>2*1E18) random number generator of L'Ecuyer with  }
{ Bays-Durham shuffle and added safeguards. Returns a uniform     }
{ deviate between 0.0 and 1.0 (exclusive of the endpoint values). }
{ Schrage's method is used to avoid intermediate overflows.       }
{ See W.H. Press et al., "Numerical Recipes in C", second edition }
{ page 282.                                                       }
{-----------------------------------------------------------------}

VAR j, k: integer;
BEGIN
  IF start THEN BEGIN
    start:= false;
    idum2:= idum;
    FOR j:= NTAB+8 DOWNTO 1 DO BEGIN
      k:= idum DIV IQ1;
      idum:= IA1*(idum-k*IQ1)-k*IR1;
      IF idum < 0 THEN idum:= idum + IM1;
      IF j <= NTAB THEN iv[j]:= idum;
    END;
    iy:= iv[1];
  END; { if }
  k:= idum DIV IQ1;
  idum:= IA1*(idum-k*IQ1)-k*IR1;
  IF idum < 0 THEN idum:= idum + IM1;
  k:= idum2 DIV IQ2;
  idum2:= IA2*(idum2-k*IQ2)-k*IR2;
  IF idum2 < 0 THEN idum2:= idum2 + IM2;
  j:= 1 + iy DIV NDIV;
  iy:= iv[j]-idum2;
  iv[j]:= idum;
  IF iy < 1 THEN iy:= iy + IMM1;
  Rand:= min(AM*iy, RNMX);
END;

{-----------------------------------------------------------------}
{ Initialize the long period random number generator -------------}
{-----------------------------------------------------------------}
GLOBAL PROCEDURE Seed(seed: integer);
BEGIN
  idum:= abs(seed);
  IF idum = 0 THEN idum:= 123;   { Avoid initial seed 0 }
  start:= true;                  { Starting procedure is active }
END;


VAR k: integer;
BEGIN
{----- Initialization of random number generator Rand ------------}
  IM1:=2147483563; IM2:=2147483399;
  AM:=1.0/IM1;
  IMM1:=IM1-1;
  IA1:=40014; IA2:=40692;
  IQ1:=53668; IQ2:=52774;
  IR1:=12211; IR2:=3791;
  NDIV:=1+IMM1 DIV NTAB;
  RNMX:= pred(1.0);
  FOR k:= 1 TO NTAB DO iv[k]:= 0;
  IDUM := 123456789;
```

```
  IDUM2:= 123456789;
  IY:= 0;
  start:= true;
END.
{--------------------- End of module Rand2 ----------------------}
```

The next module called `random` uses one of the previous modules `rand0`, `rand1` or `rand2` to generate its random values. Notice that the random number generators in each of this modules are called `rand` and the procedure to initialize the corresponding random number generator is always called `seed`. To choose the appropriate generator one has to activate the correct use clause at the beginning of the module `random`. By default the module `rand0` is included. So the fastest generator (the Minimal Standard generator wihout shuffling) is called.

```
MODULE random;
USE GLOBAL
{ rand0; { random number generator is used }
{-----------------------------------------------------------------}
{ "Minimal" random number generator                               }
{-----------------------------------------------------------------}

{ rand1; { alternatively }
{-----------------------------------------------------------------}
{ "Minimal" random number generator with Bays-Durham shuffle      }
{-----------------------------------------------------------------}

  rand2; { alternatively }
{----------------------------------------------------------------}
{ Long period (>2*1E18) random number generator of L'Ecuyer with }
{ Bays-Durham shuffle and added safeguards.                      }
{----------------------------------------------------------------}

USE i_ari, c_ari, ci_ari;

GLOBAL FUNCTION Rand(a, b: real): real;
{ Compute random numbers uniformly distributed in the range a..b }
BEGIN
  Rand:= a + (b-a)*Rand;
END;

GLOBAL FUNCTION RandLn(x: real): real;
{ Compute random numbers logarithmically distributed over (1, exp(x)) }
BEGIN
  RandLn:= exp(x*rand);
END;

GLOBAL FUNCTION RandLn(a, b: real): real;
{ Compute random numbers logarithmically distributed over (a, b) }
BEGIN
  RandLn:= a*RandLn( ln(b/a) );
END;

GLOBAL FUNCTION Rand(x: interval): interval;
{ Computes randomly a subinterval of the interval x }
VAR a, b: real;
BEGIN
  a:= Rand(inf(x), sup(x));
  b:= Rand(inf(x), sup(x));
  IF a <= b THEN
    Rand:= intval(a, b)
  ELSE
```

```
    Rand:= intval(b, a);
END;

GLOBAL FUNCTION RandLn(x: interval): interval;
{ Using logarithmically distributed random numbers subintervals }
{ of the interval x are computed }
VAR a, b: real;
BEGIN
  a:= RandLn(inf(x), sup(x));
  b:= RandLn(inf(x), sup(x));
  IF a <= b THEN
    RandLn:= intval(a, b)
  ELSE
    RandLn:= intval(b, a);
END;

GLOBAL FUNCTION Rand(u, v: complex): complex;
{ Computes randomly complex numbers with a real part in the range }
{ Re(u).. Re(v) and imaginary part in the range Im(u)..Im(v).     }
VAR
  a, b: real;
  r: complex;
BEGIN
  r.re:= Rand( u.re, v.re );
  r.im:= Rand( u.im, v.im );
  Rand:= r;
END;


GLOBAL FUNCTION Rand(z: cinterval): cinterval;
{ Computes randomly a subinterval of the complex interval z }
VAR x, y: interval;
BEGIN
  x:= Rand( z.re );
  y:= Rand( z.im );
  Rand:= compl(x, y);
END;

BEGIN  { Nothing to initialize }
END.
```

We now list a driver program which calls the random number generator chosen by the activated use clause in the module `random`. The program generates random numbers in a specified range, subintervals with randomly generated bounds and complex numbers with randomly generated real and imaginary parts. Uniformly as well as logarithmically distributed numbers are generated.

```
PROGRAM RanTst;
{ Driver program for random number generators }
USE random;

USE i_ari, c_ari;
VAR
  x, RandomInterval: interval;
  k, kmax: integer;
  u, v: complex;

BEGIN
  kMax:= 10;  { Upper bound of loops }
  seed(35);   { Initialization of the random number generator }

  writeln('Uniformly distributed random numbers in the range -41..20:');
```

```
  FOR k:= 1 TO kMax DO BEGIN
    writeln( Rand(-41,20):10:3);
  END;

  writeln;
  writeln('Uniformly distributed complex random numbers: ');
  u:= compl(-1,11); v:= compl(2,12);
  writeln('u: ', u);
  writeln('v: ', v);
  { Generate complex random numbers with real part in the range }
  {   Re(u)..Re(v) = -1..2  and imaginary part in the range     }
  {   Im(u)..Im(v) = 11..12.                                    }
  FOR k:= 1 TO kMax DO BEGIN
    writeln( Rand(u, v));
  END;

  writeln;
  writeln('Log. distributed random numbers in the range 0.5..99999:');
  FOR k:= 1 TO kMax DO BEGIN
    writeln( RandLn(0.5, 99999):10:3);
  END;

  writeln;
  x:= intval(-855.9 , 909);  { Subintervals of x will be generated }
  writeln('Subintervals of [', x.inf:10:3, ', ', x.sup:10:3, ']:' );
  FOR k:= 1 TO kMax DO BEGIN
    RandomInterval:= Rand(x);
    writeln('[',inf(RandomInterval):10:3, ', ',
                sup(RandomInterval):10:3, ']');
  END;

  writeln;
  x:= intval(0.0001 , 36909);  { Subintervals of x will be generated }
  writeln('Subintervals of [', x.inf:15:7, ', ', x.sup:15:7, ']' );
  writeln('using logarithmically distributed random numbers:' );
  FOR k:= 1 TO kMax DO BEGIN
    RandomInterval:= RandLn(x);
    writeln('[',inf(RandomInterval):12:5, ', ',
                sup(RandomInterval):12:5, ']');
  END;
END.
```

8.4 Examples

The output produced by the driver program `rantst` printed in the last section is as follows:

```
Uniformly distributed random numbers in the range -41..20:
     1.706
    -0.636
   -28.760
   -15.542
   -31.141
    -4.512
    -8.322
    -6.890
   -38.172
```

```
    15.337

Uniformly distributed complex random numbers:
u: ( -1.000000000000000E+000,  1.100000000000000E+001 )
v: (  2.000000000000000E+000,  1.200000000000000E+001 )
(  1.770523358366678E+000,  1.169023084904515E+001 )
(  8.778347934670547E-001,  1.154022708252040E+001 )
(  1.939433211857370E+000,  1.198853675929160E+001 )
(  1.795574966642946E+000,  1.145750058576816E+001 )
( -5.178980422305565E-001,  1.140738764993285E+001 )
( -1.478752338184952E-001,  1.162710895682883E+001 )
(  7.538460349090923E-001,  1.181320775585373E+001 )
( -5.881318626884410E-001,  1.199802013991015E+001 )
( -4.289019929509001E-001,  1.171562274258022E+001 )
(  7.133937462412141E-001,  1.124853307154277E+001 )

Log. distributed random numbers in the range 0.5..99999:
   223.390
    12.551
  2573.088
  4264.275
    52.079
 72772.747
    38.466
  3183.301
   302.158
     4.189

Subintervals of [  -855.900,    909.000]:
[  -847.773,    551.463]
[  -486.502,    244.339]
[  -804.325,    -46.622]
[  -752.274,   -164.046]
[   611.880,    689.147]
[  -664.705,    140.708]
[  -785.153,   -338.563]
[   242.264,    304.458]
[  -219.898,    227.582]
[  -702.911,   -318.543]

Subintervals of [      0.0001000,   36909.0000000]
using logarithmically distributed random numbers:
[      0.33334,    123.12485]
[      0.00220,     27.37289]
[      1.84284,      8.46199]
[      0.00020,      2.04612]
[      0.00012,    130.09392]
[      0.00050,      8.80838]
[      1.88097,    532.90419]
[      0.00864,     14.80861]
[      0.08226,      1.97266]
[     10.70499,     10.83823]
```

8.5 Exercises

Exercise Generate a sparse matrix A and a right hand side vector b with entries being uniformly distributed random numbers. Solve the system $A \cdot x = b$ using the sparse linear system solver discussed in Chapter 2.

8.6 Refernces and Further Readings

Two fundamental sources are [186] and [265]. Some original work may be found in [225, 264, 298].

Chapter 9

Composition of Interval Functions

9.1 Introduction

In the following paragraphs we will present different methods for the computation of interval functions. Here an interval function means the extension of a real valued function $\varphi : D \subset \mathbb{R} \longrightarrow \mathbb{R}$ which is continuous on every closed interval in its domain D to interval arguments $[x] \subset D$ by the definition

$$\varphi([x]) := \{\varphi(x) \mid x \in [x]\}.$$

In general it is not possible to compute the exact range of φ over $[x]$. But it is guaranteed that the computed range contains $\varphi(x)$ for all $x \in [x]$. Of course the goal is a sharp enclosure of the exact range of function values.

Many elementary mathematical functions are monotone at least in some subranges of D. If we are able to enclose $\varphi(x)$ for point input data $x \in D$ we can construct an enclosure of $\varphi([x]) = \varphi([\underline{x}, \overline{x}])$ using enclosures of $\varphi(\underline{x})$ and $\varphi(\overline{x})$. For example $\exp([x]) = [\exp(\underline{x}), \exp(\overline{x})]$ and $\operatorname{arcoth}([x]) = [\operatorname{arcoth}(\overline{x}), \operatorname{arcoth}(\underline{x})]$. So we are faced with the basic problem how to compute strict lower and upper bounds of the value of $\varphi(x)$ at points x. The methods we will describe here are more or less based on standard argument reduction techniques and Taylor polynomial evaluations. We also give an implementation of an accurate table lookup agorithm.

The most simple method to get an enclosure of point values will be the computation of Taylor polynomials (partial sums of series expansions) in interval arithmetic. In this way roundoff errors are bounded rigorously. Similarly, the argument reduction, though applied to thin intervals (points), is carried out in interval arithmetic. Adding to the enclosure of the Taylor polynomial the (in general small) approximation error term and performing the result adaptation again in interval arithmetic yields rigorous bounds for $\varphi(u)$, $u \in \{\underline{x}, \overline{x}\}$.

The method described so far has several disadvantages:

- due to interval computations the runtime of such routines is not optimal.
- the degrees of the Taylor polynomials are in general not as low as the degree of corresponding polynomials coming from Chebyshev approximations or even being (almost) best approximations.
- accurate argument reduction techniques using scalar products which can be computed in PASCAL-XSC by only one rounding may improve the numerical quality of the computed enclosures significantly.

The main reason for two different implementations is the significant loss of performance in case of a posteriori error estimations as compared with implementations using a priori error estimations (see the discussion for the exponential routine $\exp(x)$).

On the other hand, the implementation using a posteriori error estimations can be applied to other *real* data formats including multiple presicion formats (see Section 7.4.6 and [190]) whereas the implementation using a priori error estimations is restricted to the *real* format as specified in Section 9.2 . In the subsequent paragraphs we will discuss these problems in more detail.

For functions which are not monotone additional considerations have to be applied. As examples, we will discuss the sine function as a periodic function with an irrational period length and the power function as a function depending on two arguments. The example of the interval power function is mainly intended to give an expression of what has to be done to handle real and imaginary parts (they can be interpreted as functions of two real variables) of complex interval functions. For the implementation of complex interval functions with high accuracy refer to [68], [70] and [188].

Before we treat the implementation of (interval) functions in more detail we discuss the underlying real data format of PASCAL-XSC.

9.2 PASCAL-XSC Data Type *real*

The material discussed in this section is mainly taken from the PASCAL-XSC User's Guide [262].

The PASCAL–XSC data type *real* consists of all floating-point numbers and special values which are specified by the IEEE standard [143] for the double floating-point number format. A variable of type *real* is 64 bits long and requires eight bytes of storage. In Figure 9.1, a sketch of the IEEE double floating-point data format is given.

The largest and the smallest positive real value representable by the data type *real* are denoted by *maxreal* and *minreal*, respectively. The constants *maxreal* and *minreal* are defined in module `x_real`.

The floating-point number format uses a binary representation (base B=2) of the mantissa digits. There is one implicitly defined hidden bit in the representation of normalized and denormalized numbers, thus making a total length of the mantissa m of 53 bits. The exponent field e occupies 11 bits. For normalized floating-point numbers according to the notation in Figure 9.1, the range of exponent values is specified by the maximum exponent $e_{max} = 2046 - 1023 = 1023$ and the minimum exponent $e_{min} = 1 - 1023 = -1022$. The sign field s occupies 1 bit.

Example The decimal number 0.1 can be written in the form

$$0.1 = \sum_{k=1}^{\infty} (2^{-4k} + 2^{-(4k+1)}) \ .$$

	1	11	52	bit count
(r)	s	e	m	
	63	62 52	51 0	bit number

$$
\mathrm{r} = \begin{cases}
(-1)^s \cdot 1.m \cdot 2^{e-1023} & 1 \le e \le 2046 \quad \text{normalized} \\
(-1)^s \cdot 0.m \cdot 2^{-1022} & e = 0, m \ne 0 \quad \text{denormalized} \\
(-1)^s \cdot 0 & e = 0, m = 0 \quad \text{signed zero} \\
(-1)^s \cdot \infty & e = 2047, m = 0 \quad \text{signed infinity} \\
\text{NaN} & e = 2047, m \ne 0 \quad \text{not a number}
\end{cases}
$$

Figure 9.1: IEEE double floating-point format

Hexadecimal Representation	Decimal Value
`0000000000000001` *minreal*	$4.9406564584124654 \cdot 10^{-324}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields 0.0.
`7FEFFFFFFFFFFFFF` *maxreal*	$1.7976931348623158 \cdot 10^{308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields $+\infty$.

Figure 9.2: *real* constants *minreal* and *maxreal*

Hexadecimal Representation	Rounding Mode	Decimal Value
`3FF0000000000000`	exact	1.0
`BFF0000000000000`	exact	-1.0
`3FE0000000000000`	exact	0.5
`3FB999999999999A`	$\square$	0.1
`3FB9999999999999`	$\bigtriangledown$	0.1
`3FB999999999999A`	$\bigtriangleup$	0.1
`400921FB54442D18`	$\square$	π
`3FF921FB54442D18`	$\square$	$\pi/2$

Figure 9.3: Some decimal values and their IEEE representation

Thus, the value 0.1 is not exactly representable in any binary floating-point system. Its *infinite* binary representation is $0.000\overline{1100}_2$. In Figure 9.3 the IEEE representations (with hidden bit technique) for different rounding modes are shown.
Special values named "not a number" (NaN) may be signaling or quiet (see Appendix B for the classification of `real` numbers and the IEEE exception handling).
Representations of some special values of the *real* data format are listed in Figure 9.4.

The order in which the bytes of a *real* value are stored depend on the storage conventions used by the hardware. This is most important in those cases where hardware support for arithmetic operations is used. Refer to the *local configuration guide* for details.
Exercise Give the binary representation of 1/7. What is its IEEE double representation? What is the relative error of $\square(1/7)$?

9.3 Module *x_real*

For the purpose of showing the bits of a real number in the order indicated by Figure 9.1 we can use the PASCAL-XSC standard module `x_real`.
Module `x_real` contains additional constants, types, functions, and procedures for an extended or alternative processing of *real* values. For the classification of *real* values we refer to Appendix B.

9.3.1 Composition and Decomposition of *real* Values

The default functions *comp*, *expo*, and *mant* assume an abstract representation of a non-zero normalized floating-point number with a normalized mantissa m satisfying $B^{-1} \le |m| < 1$. Here, B stands for the base which is used for the representation of the digits of mantissa m.
In the special case $B = 2$, the abstract representation of a non-zero normalized floating-point number with a mantissa m satisfying $1 \le |m| < 2$ is possible, too. The binary IEEE data formats are defined by such a formulation. The functions `x_comp`, `x_expo` and `x_mant` handle *real* values according to this abstract representation.

```
function x_comp ( m : real; e : integer ) : real;
function x_expo ( r : real ) : integer;
function x_mant ( r : real ) : real;
```

9.3.2 Formatted Input/Output for *real* Values

The representation of *real* input and output values in a hexadecimal notation is made available by overloading the procedures *read* and *write* for *real* arguments.

```
procedure read ( var f : text; var r : real; mode : char );
procedure write( var f : text;     r : real; mode : char );
```

Hexadecimal representation	Decimal value
`FFF0000000000000`	$-\infty$
`FFEFFFFFFFFFFFFF`	$-1.7976931348623158 \cdot 10^{308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields $-\infty$.
`BFF0000000000000`	-1.0
`8010000000000000`	$-2.2250738585072013 \cdot 10^{-308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields the smallest denormalized number.
`8000000000000001`	$-4.9406564584124654 \cdot 10^{-324}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields -0.0.
`8000000000000000`	-0.0
`0000000000000000`	0.0
`0000000000000001` *minreal*	$4.9406564584124654 \cdot 10^{-324}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields 0.0.
`0010000000000000`	$2.2250738585072013 \cdot 10^{-308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields the largest denormalized number.
`3FF0000000000000`	1.0
`7FEFFFFFFFFFFFFF` *maxreal*	$1.7976931348623158 \cdot 10^{308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields $+\infty$.
`7FF0000000000000`	$+\infty$

Figure 9.4: Special *real* values

Both the input format and the output format is a fixed-format 16 digit hexadecimal notation. The left-most hexadecimal digit holds bit 63 through bit 60 of the floating-point number format as specified in Figure 9.1. The right-most hexadecimal digit holds bit 3 through bit 0.
Procedure `read` for hexadecimal input does not distinguish between lower case hexadecimal digits `a`, `b`, `c`, `d`, `e`, `f` and upper case hexadecimal digits `A`, `B`, `C`, `D`, `E`, `F`. Possible `mode` characters are '`x`' and '`X`'.
Procedure `write` may be called with `mode` character '`x`' or '`X`', which will produce lower case hexadecimal digits `a`, `b`, `c`, `d`, `e`, `f` or upper case hexadecimal digits `A`, `B`, `C`, `D`, `E`, `F`, respectively.
The PASCAL–XSC statement

```
writeln(1.0:'x',' = ',1.0:'X')
```

yields the following output line:

```
3ff0000000000000 = 3FF0000000000000
```

9.4 Measure of Errors

As usual we introduce for a nonzero number x and its approximation $\tilde{x}$ the following two measures of error:

$$\varepsilon(\tilde{x}) := \frac{\tilde{x} - x}{x} \tag{9.1}$$

and

$$\Delta(\tilde{x}) := \tilde{x} - x \tag{9.2}$$

It holds

$$\tilde{x} = (1 + \varepsilon(\tilde{x})) \cdot x \tag{9.3}$$

The following relative error bound for the machine result $a \boxdot b$ is fulfilled by all four basic arithmetical operations $\circ \in \{+, -, \cdot, /\}$ whenever the corresponding exact result $a \circ b$ lies in the range of normalized numbers:

$$\left| \frac{a \circ b - a \boxdot b}{a \circ b} \right| < \varepsilon := 2^{-53} \tag{9.4}$$

Here a, b denote normalized IEEE double numbers (see Figure 9.1). Notice, that for denormalized numbers this error bound is in general no longer valid. The so called machine epsilon (Wilkinson's epsilon) is defined by

$$\varepsilon := 0.5 \cdot B^{1-l} = 2^{-53} \in [1.11022E - 16,\ 1.11023E - 16] \tag{9.5}$$

where $l = 53$ is the number of digits with respect to the basis $B = 2$ of the IEEE double floating-point system.

9.5 Sine Function

We now consider the realization of mathematical functions. Let us first discuss a technique for bounding the sine for point arguments.

9.5.1 Sine Function for Point Arguments

We develop a very simple technique to get an enclosure of $\sin(x)$ for a point argument x. For the computation of $\sin(x)$ the reduction of the argument into the so called basic range (a small region with center zero) plays an important role.

9.5.1.1 Argument Reduction

For the given argument x an argument reduction is performed into the range $[-\frac{\pi}{4}, \frac{\pi}{4}]$. The reduced argument t can be computed by

$$\begin{array}{rcll} k & := & x \cdot \dfrac{2}{\pi} & \text{(real value)} \\ k & := & \text{round}(k) & \text{(integer value)} \\ t & := & x - k \cdot \dfrac{\pi}{2} & \in [-\frac{\pi}{4}, \frac{\pi}{4}] \end{array} \tag{9.6}$$

The reduction is illustrated in Figures 9.5 and 9.6. In Figure 9.6 segments plotted using the same line style are all mapped to their corresponding basic curve segment which crosses the y-axis.

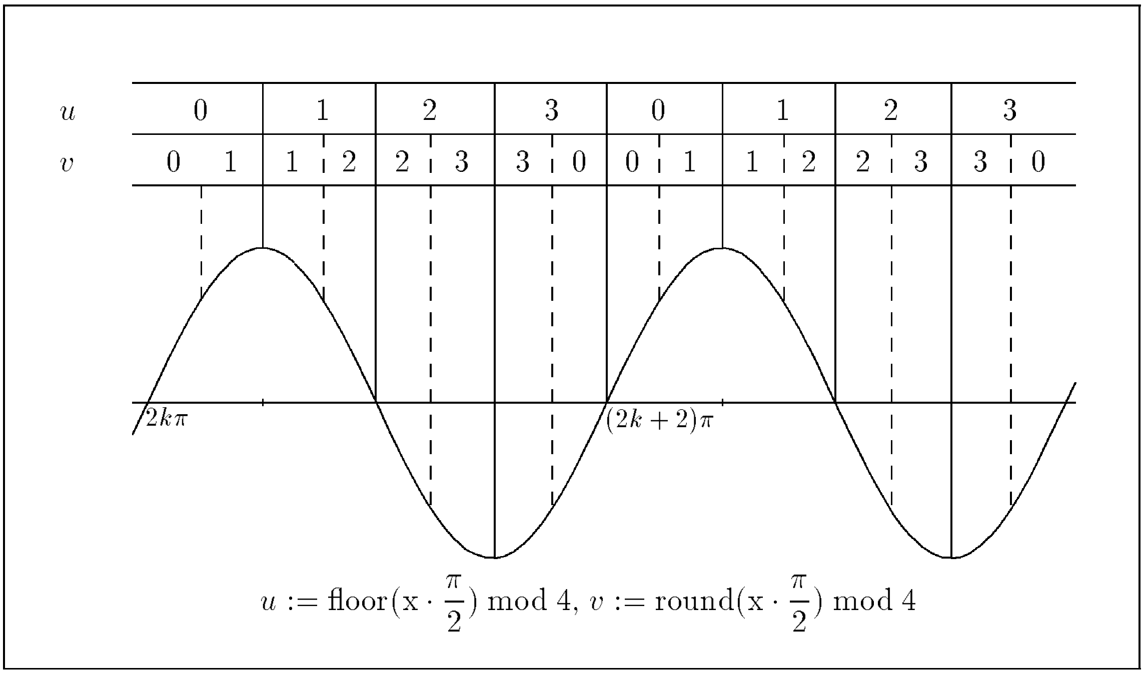


Figure 9.5: Reduction Process for the Sine Function

Using this reduction (it results in the two values k and t) $\sin(x)$ can be computed according to

$$\sin(x) = \sin(t, k \bmod 4) \tag{9.7}$$

where the function $\sin(t, m)$ is defined by

$$\sin(t,m) := \begin{cases} \sin(t) & \text{if} \quad m = 0 \\ \cos(t) & \text{if} \quad m = 1 \\ -\sin(t) & \text{if} \quad m = 2 \\ -\cos(t) & \text{if} \quad m = 3 \end{cases} \tag{9.8}$$

Relation (9.8) is illustrated by Figure 9.6.

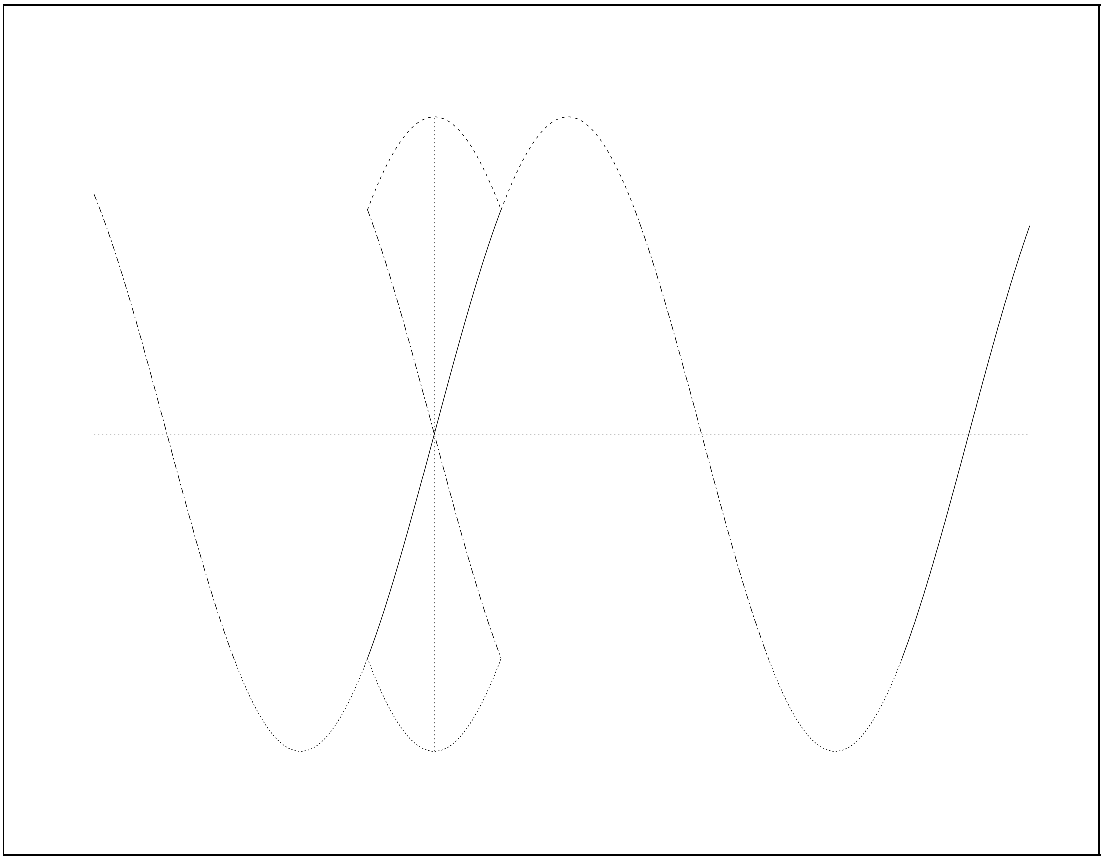


Figure 9.6: Reduction for the Sine to the Basic Range $[-\frac{\pi}{4},\ \frac{\pi}{4}]$

9.5.1.2 Approximation within the Basic Range

Now $\sin(t)$ and $\cos(t)$ for $t \in [-\frac{\pi}{4}, \frac{\pi}{4}]$ can be computed using their taylor expansions

$$\sin(t) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} t^{2k+1} =: t \cdot p_\infty(t^2), \tag{9.9}$$

$$\cos(t) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} t^{2k} =: q_\infty(t^2) \tag{9.10}$$

If we use the approximation $t \cdot p_n(t^2) := \sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)!} t^{2k+1} \approx \sin(t)$ the approximation error is bounded by

$$\begin{aligned}
\left|\frac{\sin(t) - t \cdot p_n(t^2)}{\sin(t)}\right| &= \left|\frac{t \cdot \sum_{k=n+1}^{\infty} \frac{(-1)^k}{(2k+1)!} t^{2k}}{t \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} t^{2k}}\right| \\
&\le \frac{1}{1 - \frac{1}{6}t^2} \cdot \frac{t^{2n+2}}{(2n+3)!} \underbrace{\left|1 - \underbrace{\frac{(2n+3)!}{(2n+5)!}}_{<1} t^2 + \underbrace{\frac{(2n+3)!}{(2n+7)!}}_{<1} t^4 - + \ldots\right|}_{\le 1} \\
&\le \frac{1}{1 - \frac{1}{6}t^2} \cdot \frac{t^{2n+2}}{(2n+3)!}
\end{aligned} \tag{9.11}$$

Thus, for $t \in [-\frac{\pi}{4}, \frac{\pi}{4}]$ the relative approximation error is bounded by

$$\left|\frac{\sin(t) - t \cdot p_n(t^2)}{\sin(t)}\right| \le \frac{1}{1 - \frac{1}{6}\left(\frac{\pi}{4}\right)^2} \cdot \frac{\left(\frac{\pi}{4}\right)^{2n+2}}{(2n+3)!} =: \varepsilon(\sin, n). \tag{9.12}$$

For the approximation error of $q_n(t) := \sum_{k=0}^{n} \frac{(-1)^k}{(2k)!} t^{2k} \approx \cos(t)$ we find

$$\begin{aligned}
\left|\frac{\cos(t) - q_n(t^2)}{\cos(t)}\right| &\le \left|\frac{\sum_{k=n+1}^{\infty} \frac{(-1)^k}{(2k)!} t^{2k}}{1 - \frac{1}{2}t^2}\right| \\
&\le \frac{1}{1 - \frac{1}{2}t^2} \cdot \frac{t^{2n+2}}{(2n+2)!} \underbrace{\left|1 - \frac{(2n+2)!}{(2n+4)!} t^2 + \frac{(2n+2)!}{(2n+6)!} t^4 - + \ldots\right|}_{<1} \\
&\le \frac{1}{1 - \frac{1}{2}\left(\frac{\pi}{4}\right)^2} \cdot \frac{\left(\frac{\pi}{4}\right)^{2n+2}}{(2n+2)!} =: \varepsilon(\cos, n) \text{ for all } t \in [-\frac{\pi}{4}, \frac{\pi}{4}].
\end{aligned} \tag{9.13}$$

n	$\varepsilon(\sin, n)$	$\varepsilon(\cos, n)$
5	9.8608205E-012	1.6630390E-010
6	2.8965000E-014	5.6365166E-013
7	6.5687749E-017	1.4487029E-015
8	1.1847809E-019	2.9203685E-018
9	1.7400772E-022	4.7406055E-021

Table 9.1: Errors for Sine and Cosine for Approximations with Different Degrees

Table 9.1 shows some values of $\varepsilon(\sin, n)$ and $\varepsilon(\cos, n)$ for different values of n. We can see that for the sine $n := 7$ and for cosine $n := 8$ lead to error bounds that are appropriate for the IEEE double format (64-bit, see Figure 9.1 and (9.5)). The machine epsilon for this fomat is $\varepsilon = 2^{-53} = 1.110223\ldots 10^{-16}$.

To get an enclosure of $\sin(t)$, $t \in [-\frac{\pi}{4}, \frac{\pi}{4}]$ we proceed as follows. Let $a_k := \dfrac{(-1)^k}{(2k+1)!}$, $b_k := \dfrac{(-1)^k}{(2k)!}$, $k = 0, \ldots, n$ denote the polynomial coefficients of $p_n(t) \approx \sin(t)$ and $q_n(t) \approx \cos(t)$ and $[a_k]$ and $[b_k]$ be intervals with $a_k \in [a_k], b_k \in [b_k], k = 0, 1, \ldots, n$. Using the inclusion monotonicity of interval arithmetic we get with $[tt] := t \diamond t \in I\!\!R$

$$\begin{aligned} t \cdot p_n(t^2) \quad \in \quad & t \diamond (\ldots(([a]_n \diamond [tt] \oplus [a]_{n-1}) \diamond [tt] \oplus [a]_{n-2}) \\ & \diamond [tt] \oplus \ldots) \oplus [tt] \oplus [a]_0 =: [t \cdot p_n(t^2)] \end{aligned} \tag{9.14}$$

and using the error bound $\varepsilon(\sin, n) =: \varepsilon(\sin)$,

$$\sin(t) \in [t \cdot p_n(t^2)] \diamond [1 - \varepsilon(\sin), 1 + \varepsilon(\sin)] =: [s(t)].$$

Similarly one finds

$$\cos(t) \in [q_n(t^2)] \diamond [1 - \varepsilon(\cos), 1 + \varepsilon(\cos)] =: [c(t)]\,. \tag{9.15}$$

Here $[q_n(t^2)]$ denotes the interval Horner scheme corresponding to the polynomial $q_n(t^2)$ and $\varepsilon(\cos) := \varepsilon(\cos, n)$. Up to now we are able to compute enclosures of $\sin(t)$ and $\cos(t)$ for point arguments lying in the range $[-\frac{\pi}{4}, \frac{\pi}{4}]$ (reduced argument range). The interval functions $[s(t)]$ and $[c(t)]$ can be combined to get an interval analogon $\mathrm{Sin}([t], m)$ to $\sin(t, m)$ as defined in (9.8):

$$\mathrm{Sin}([t], m) := \begin{cases} [s([t])] & \text{if} \quad m = 0 \\ [c([t])] & \text{if} \quad m = 1 \\ -[s([t])] & \text{if} \quad m = 2 \\ -[c([t])] & \text{if} \quad m = 3 \end{cases} \tag{9.16}$$

Here $[t]$ denotes an interval wich contains the reduced argument t, i.e. $[t]$ may be the result of an argument reduction according to (9.6), which is performed in interval arithmetic:

$$\begin{array}{rcl} k & := & x \boxdot \left.\dfrac{2}{\pi}\right|_{\text{IEEE}} \\ k & := & \text{round}(k) \\ [t] & := & x \diamondminus k \diamonddot \left[\dfrac{\pi}{2}\right] \in I\!\!R \end{array} \quad . \tag{9.17}$$

The quantity $\left.\dfrac{2}{\pi}\right|_{\text{IEEE}}$ is the IEEE double floating-point number nearest to the real value $2/\pi$ and $\left[\dfrac{\pi}{2}\right]$ denotes the smallest intervall with IEEE double bounds which contains $\pi/2$. Then, using relations (9.17) and (9.7), we find the enclosure

$$\sin(x) = \sin(t,\ k \bmod 4) \in \text{Sin}([t], k \bmod 4) \in I\!\!R\,. \tag{9.18}$$

9.5.2 Sine Function for Interval Arguments

In the previous paragraph we have developed an algorithm to compute an enclosure of $\sin(x)$ for an arbitrary point argument $x \in I\!\!R$. Let us now discuss the intervall case. Again we are interested in a simple method, which avoids a priori error estimations.
To compute an enclosure of $\sin([\,\underline{x}, \overline{x}\,])$ we proceed as follows:

1. reduce the point argument $\underline{x}$ to get $[\,\underline{t}\,]$ and $\underline{k}$ according to (9.17).

2. reduce $\overline{x}$ to get $[\,\overline{t}\,]$ and $\overline{k}$ according to (9.17).

3. set the boolean values r_{lb}, r_{ub} as follows:

$$r_{lb} := [x] \cap \{\ldots,\ -\frac{5}{2}\pi,\ -\frac{1}{2}\pi,\ \frac{3}{2}\pi,\ \frac{7}{2}\pi,\ \frac{11}{2}\pi, \ldots\} = \emptyset$$

$$r_{ub} := [x] \cap \{\ldots,\ -\frac{3}{2}\pi,\ \frac{1}{2}\pi,\ \frac{5}{2}\pi,\ \frac{9}{2}\pi, \ldots\} = \emptyset$$

   i.e. r_{lb} is true, if the interval $[x] = [\,\underline{x}, \overline{x}\,]$ under consideration contains a real number, say y, with $\sin(y) = -1$ and r_{ub} is true, if $[x]$ contains a value z, with $\sin(z) = 1$.

4. $$[u] := \text{iSin}([\,\underline{t}\,], \underline{k} \bmod 4) \quad (\ni \sin(\underline{x}))$$

   $$[v] := \text{iSin}([\,\overline{t}\,], \overline{k} \bmod 4) \quad (\ni \sin(\overline{x}))$$

5. **if** r_{lb} **then**
   $\underline{r} := -1$
   else
   $\underline{r} := \min\{\inf[u], \inf[v]\}$
   if r_{ub} **then**
   $\overline{r} := 1$
   else
   $\overline{r} := \max\{\sup[u], \sup[v]\}$

Now it holds

$$\sin([\,\underline{x}, \overline{x}\,]) \subseteq [\,\underline{r}, \overline{r}\,]$$

9.5.2.1 Algorithms

The argument reduction for the sine is done according to

Algorithm 9.1: ArgRed $(x, [t], k)$ {subroutine}

1. $k := \text{round}\left(x \boxdot \left.\frac{2}{\pi}\right|_{\text{IEEE}}\right)$ { floating point computation }
2. $[t] := x \diamondminus k \diamonddot \left[\frac{\pi}{2}\right]$ { reduced argument }
3. **return** $[t], k$

Compute an enclosure of $\sin(t)$ with t lying in the basic range as follows:

Algorithm 9.2: Sin $([t])$ {function}

1. $[a_k] := \left[\frac{(-1)^k}{(2k+1)!}\right]$, $k = 0, 1, \ldots, n := 7$, $\varepsilon(\sin) := \varepsilon(\sin, n)$ (see Table 9.1)
2. $[tt] := [t][t]$
3. $[s] := [a_n]$ {Interval Horner's scheme}
   for $j := n-1$ **downto** 0 **do**
   $[s] := [s][tt] + [a_j]$
4. $[s] := [t][s][1-\varepsilon(\sin), 1+\varepsilon(\sin)]$ {Take into account the approximation error}
5. **return** Sin $:= [s]$

Now we list the corresponding algorithm for enclosing $\cos(t)$ with t in the basic interval:

Algorithm 9.3: $\mathsf{Cos}\,([t])$ {function}

1. $[b_k] := [\dfrac{(-1)^k}{(2k)!}], k = 0, 1, \ldots, n := 8, \quad \varepsilon(\cos) := \varepsilon(\cos, n)$ (see Table 9.1)
2. $[tt] := [t][t]$
3. $[s] := [b_n]$
   for $j := n - 1$ **downto** 0 **do**
   $\qquad [s] := [s][tt] + [b_j]$
4. $[s] := [s][1 - \varepsilon(\cos), 1 + \varepsilon(\cos)]$ {Take into account the approximation error}
5. **return** $\mathsf{Cos} := [s]$

Algorithm 9.4: $\mathsf{Sin}\,([t], m)$ {function}

1. **if** $m = 0$ **then**
   $\qquad [s] := \mathsf{Sin}\,([t])$
   else if $m = 1$ **then**
   $\qquad [s] := \mathsf{Cos}\,([t])$
   else if $m = 2$ **then**
   $\qquad [s] := -\mathsf{Sin}\,[t]$
   else
   $\qquad [s] := -\mathsf{Cos}\,[t]$
2. **return** $\mathsf{Sin} := [s]$

If we only want to enclose function values for point arguments, we can use

Algorithm 9.5: $\mathsf{SinSimp}\,(x)$ {function}

1. Compute $[t]$ and k by $\mathsf{ArgRed}\,(x, [t], k)$
2. $[s] := \mathsf{Sin}\,([t], k \bmod 4)$
3. **return** $\mathsf{SinSimp} := [s]$

An enclosure of $\sin[x]$ for an interval argument $[x]$ is computed by

Algorithm 9.6: $\mathsf{SinSimp}\,([x])$ {function}

1. Compute $[\,\underline{t}\,], \underline{k}$ by $\mathsf{ArgRed}\,(\underline{x}, [\,\underline{t}\,], \underline{k})$
2. Compute $[\,\overline{t}\,], \overline{k}$ by $\mathsf{ArgRed}\,(\overline{x}, [\,\overline{t}\,], \overline{k})$
3. $r_{lk} := [x] \cap \{\frac{3}{2}\pi \pm j2\pi | j = 0, 1, 2 \ldots\} = \emptyset$

4. $r_{ub} := [x] \cap \{\frac{1}{2}\pi \pm j2\pi | j = 0, 1, 2, \ldots\} = \emptyset$
5. $[u] := \mathsf{Sin}\,([\,\underline{t}\,], \underline{k} \bmod 4)$

   $[v] := \mathsf{Sin}\,([\,\overline{t}\,], \overline{k} \bmod 4)$
6. **if** r_{lb} **then** $\underline{r} := -1$

   else $\underline{r} := \min\{\inf[u], \inf[v]\}$
7. **if** r_{ub} **then** $\overline{r} := 1$

   else $\overline{r} := \max\{\sup[u], \sup[v]\}$
8. **return** SinSimp $:= [\,\underline{r}, \overline{r}\,]$

9.5.2.2 Program Code

The module `sinm` supplies all functions and subroutines as introduced in the last paragraph. Only the routine SinSimp is defined as a global function. It allows to compute the range of the sine over an interval $[x]$. We do not redefine the PASCAL-XSC intrinsic function `sin`. So our test routines can use the new algorithm as well as the intrinsic PASCAL-XSC version. The implementation given here is far away from being optimal. If one uses a priori error estimations including all possible rounding errors and errors introduced by the conversion of real constants into the floating point screen the algorithm will run at least as twice as fast. The efficiency can be even more increased even more using best approximations for sine and cosine within the basic interval $[-\frac{\pi}{4}, \frac{\pi}{4}]$. The numerical quality can also be increased significantly. See the remarks following our numerical examples.
We now list the program code according to the algorithms given in the last paragraph.

```
MODULE sinm; { Sine Modul Simple Version }

{ Enclosure of sin[x] (most simple but inefficient method) }

USE i_ari;    { Make available interval arithmetic }
USE fnc_util; { Make available Min, Max, ...       }

CONST SinEps = 1e-15; { Bound for the approximation error of sin }
CONST CosEps = 1e-15; { Bound for the approximation error of cos }

CONST SinMax= 7;       { Degree of the polynomial approximation for sin }
CONST CosMax= 8;       { Degree of the polynomial approximation for cos }
VAR a: ARRAY[0..SinMax] OF interval;  { polynomial coefficients for sin }
VAR b: ARRAY[0..CosMax] OF interval;  { polynomial coefficients for cos }

{ Variables for Sin and Cos to be initialized in the body of this module }
VAR
   SinUpDown: interval;
   CosUpDown: interval;

FUNCTION Sin(t: interval): interval;
{ Sine function for interval arguments t in the range [-pi/4, pi/4] }
{ Notice: The intrinsic sin of PASCAL-XSC is overwritten in this module }
VAR
  k: integer;
  s, tt: interval;
```

```
  { Global variables are SinMax and polynomial coefficients a[0..SinMax] }
  {                      and SinUpDown;     t in [-pi/4, pi/4 ]          }
BEGIN
  { Perform interval Horner scheme for sin }
  tt:= t*t;
  s:= a[SinMax];
  FOR k:= SinMax-1 DOWNTO 0 DO BEGIN
    s:= s*tt + a[k];
  END;
  s:= s*t;
  { Take approximation error into account }
  Sin:= s*SinUpDown;
END;

FUNCTION Cos(t: interval): interval;
{ Cosine function for interval arguments t in the range [-pi/4, pi/4]   }
{ Notice: The intrinsic cos of PASCAL-XSC is overwritten in this module }
VAR
  k: integer;
  s, tt: interval;
  { Global variables are CosMax and polynomial coefficients b[0..CosMax] }
  {                      and CosUpDown                                   }
BEGIN
  { Perform interval Horner scheme for cos }
  tt:= t*t;
  s:= b[CosMax];
  FOR k:= CosMax-1 DOWNTO 0 DO BEGIN
    s:= s*tt + b[k];
  END;
  { Take approximation error into account }
  Cos:= s*CosUpDown;
END;


FUNCTION Sin(r: interval; n: integer): interval;
{ sin(x) with x reduced to r in [-pi/4, pi/4]                     }
{   r: reduced argument                                           }
{   n: indicates whether the original argument x is in the range }
{        [-pi/4, pi/4  ]    + k*2*pi ==>  n = 0                   }
{        [ pi/4, 3*pi/4]    + k*2*pi ==>  n = 1                   }
{        [ 3*pi/4, 5*pi/4] + k*2*pi ==>  n = 2                    }
{        [ 5*pi/4, 7*pi/4] + k*2*pi ==>  n = 3                    }
{  n = round(x*2/pi) mod 4 is computed along with the argument reduction }

VAR sinx: interval;
{ Calls to Sin and Cos are calls to the corresponding functions }
{   implemented in this module                                  }
BEGIN
  IF n = 0 THEN BEGIN
    sinx:= Sin(r);
  END
  ELSE IF n = 1 THEN BEGIN
    sinx:= Cos(r);
  END
  ELSE IF n = 2 THEN BEGIN
    sinx:= -Sin(r);
  END
  ELSE  BEGIN { n = 3 }
    sinx:= -Cos(r);
  END;
  Sin:= sinx;
END;

{ Variables to be initialized for SinSimp in the body of this module }
```

```
VAR
   TwoOverPi: real;
   PiOverTwo: interval;
   MaxInt   : integer;
   SinHuge  : real;

GLOBAL FUNCTION SinSimp(x: interval): interval;
{ We do  n o t  overwrite the intrinsic sin globally. Instead we      }
{   define a new function iSin. This allows calls to the intrinsic    }
{   sine as well as calls to our new routine iSin outside this module }
{ Global variables are PiOverTwo, TwoOverPi, SinHuge                  }
VAR
  k     : integer;
  ku, ko: integer;
  lb, ub: boolean;
  u, v, r: interval;
  h: real;
BEGIN

  IF (abs(inf(x)) > SinHuge) OR (abs(sup(x)) > SinHuge) THEN BEGIN
    writeln(' Argument in iSin out of range!!! ');
  END;

  { Perform reduction for inf(x) }
  ku:= round( inf(x) * TwoOverPi );
  u:= inf(x) - ku*PiOverTwo;

  { Perform reduction for sup(x) }
  ko:= round( sup(x) * TwoOverPi );
  v:= sup(x) - ko*PiOverTwo;

  { Enclose sin(inf(x))  and sin(sup(x)) }
  u:= Sin(u, ku MOD 4);
  v:= Sin(v, ko MOD 4);

  { Test if -1 or +1 is the minimum or maximum, respectively }
  k:= ku-2;
  lb:= false; ub:= false;
  REPEAT
    k:= k+1;
    lb:= lb OR (( k MOD 4 = 3 ) AND NOT (x >< k*PiOverTwo));
    ub:= ub OR (( k MOD 4 = 1 ) AND NOT (x >< k*PiOverTwo));
  UNTIL lb AND ub  OR (k = ko+1);

  IF lb THEN BEGIN
    r.inf:= -1;
  END
  ELSE BEGIN
    r.inf:= min(inf(u), inf(v));
  END;

  IF ub THEN BEGIN
    r.sup:= 1;
  END
  ELSE BEGIN
    r.sup:= max(sup(u), sup(v));
  END;
  SinSimp:= r;
END;

{ Variables only used in the initialization part }
VAR k, sign: integer;

BEGIN  { Initialization part }
```

```
    SinUpDown:= intval( 1.0 -< SinEps, 1.0 +> SinEps );
    CosUpDown:= intval( 1.0 -< CosEps, 1.0 +> CosEps );
    a[0]:= 1;
    sign:= -1;
    FOR k:= 1 TO SinMax DO BEGIN
       a[k]:= sign*intval(1)/Factorial(2*k+1); { approximation for sin }
       sign:= -sign;
    END;
    b[0]:= 1;
    sign:= -1;
    FOR k:= 1 TO CosMax DO BEGIN
       b[k]:= sign*intval(1)/Factorial(2*k);   { approximation for cos }
       sign:= -sign;
    END;
    TwoOverPi:= 2/(4*arctan(1.0));
    PiOverTwo:= (4*arctan(intval(1.0)))/2;
    MaxInt:= 2147483647;  { 2**31 - 1 }
    SinHuge:= MaxInt*inf(PiOverTwo);
 END.
```

9.5.2.3 Numerical Examples

A simple driver program is shown now:

```
PROGRAM sinmtst;   { Driver for module sinm }

USE i_ari; { Interval operations }
USE sinm;  { Make the new interval function SinSimp available }

VAR x, iSinx, sinx: interval;
    k: integer;
BEGIN
   { Compare the intrinsic sin and iSin for some powers of two }
   FOR k := -2 TO 10 DO BEGIN
      x:= power(2.0, 3*k);    { x = 2**(3k) }
      iSinx:= SinSimp(x);  { Simple version of the sine    }
      sinx := sin(x);      { PASCAL-XSC intrinsic function }
      writeln('SinSimp(2**', 3*k:2, '): ', iSinx);
      IF NOT(sinx <= iSinx) THEN writeln('*** A t t e n t i o n  !!!');
   END;
   writeln('sin(2**30) using the intrinsic sin of PASCAL-XSC:');
   writeln('   sin: ', sinx);
   writeln('sin(2**30) using the new function SinSimp:');
   writeln('  iSin: ', iSinx);
END.
```

Running the driver program `sinmtst` produces the following output:

```
SinSimp(2**-6): [   1.56243642248833E-002,   1.56243642248834E-002 ]
SinSimp(2**-3): [   1.24674733385227E-001,   1.24674733385228E-001 ]
SinSimp(2** 0): [   8.41470984807895E-001,   8.41470984807898E-001 ]
SinSimp(2** 3): [   9.89358246623380E-001,   9.89358246623384E-001 ]
SinSimp(2** 6): [    9.2002603819678E-001,    9.2002603819680E-001 ]
SinSimp(2** 9): [      7.95184940128E-002,      7.95184940130E-002 ]
SinSimp(2**12): [     -5.94641987609E-001,     -5.94641987607E-001 ]
SinSimp(2**15): [      9.27856333412E-001,      9.27856333416E-001 ]
SinSimp(2**18): [       -8.410702783E-002,       -8.410702776E-002 ]
SinSimp(2**21): [        6.238443990E-001,        6.238443996E-001 ]
SinSimp(2**24): [        -7.79563676E-001,        -7.79563671E-001 ]
SinSimp(2**27): [         -7.6340325E-001,         -7.6340322E-001 ]
SinSimp(2**30): [          -6.173267E-001,          -6.173261E-001 ]
sin(2**30) using the intrinsic sin of PASCAL-XSC:
```

```
    sin: [ -6.173264150460423E-001, -6.173264150460421E-001 ]
 sin(2**30) using the new function SinSimp:
   iSin: [          -6.173267E-001,         -6.173261E-001 ]
```

For larger arguments (all data are point intervals) the enclosures are no longer satisfactorily. The great width of the enclosures is due to the inaccurate argument reduction.

9.5.3 Improved Version for $\sin([x])$

9.5.3.1 Accurate Argument Reduction

The realization of the argument reduction

$$k := \mathrm{round}(x \cdot \frac{2}{\pi})$$

$$t := x - k \cdot \frac{\pi}{2}$$

by naive interval arithmetic (see (9.17)) leads for arguments x near to a multiple of $\pi/2$ or for $|x|$ large to a significant loss of accuracy. The relative error may be very large. This loss of accuracy can be easily avoided using the accurate dot product of PASCAL-XSC:
Let the constants $c_j, j = 1, 2, \ldots, j_{\max}$ be defined by

$$c_1 := \triangledown(\frac{\pi}{2}) \tag{9.19}$$

$$c_k := \triangledown(\frac{\pi}{2} - \sum_{j=1}^{k-1} c_j),\ k = 2, \ldots, j_{\max} - 1 \tag{9.20}$$

$$[c_{j_{\max}}] := \Diamond(\frac{\pi}{2} - \sum_{j=1}^{j_{\max}-1} c_j). \tag{9.21}$$

The sum of the c_j is a very good approximation (about $j_{\max} \cdot 53$ bits are correct) for $\frac{\pi}{2}$ and it holds $\frac{\pi}{2} \in \sum_{j=1}^{j_{\max}-1} c + [c]_{j_{\max}}$.
Using these values, we make the following reduction

$$k := \mathrm{round}(\Box(x \boxdot \Box(\frac{2}{\pi}))) \tag{9.22}$$

$$[t] := \Diamond(\,x - k \sum_{j=1}^{j_{\max}-1} c_j - k \cdot [c_{j_{\max}}]\,) \tag{9.23}$$

The formula for $[t]$ is computed as an interval scalar product. So the exact result of the expression is rounded outward only once. The number $j_{\max}$ of constants c_j

has to be chosen appropriately. We will not do this by an a priori error estimation. Heuristically we will expect a cancellation of 53 bits for arguments very close to a small multiple of $\pi/2$ and for arguments with large absolut value ($\approx$ maxint $\cdot$ $\pi/2 \approx (2^{31}-1)\cdot 1.57 \approx 3.37E9$) additionally 30 bits may suffer by cancellation. An approximation to $\pi/2$ by the sum of 3 IEEE numbers should produce reduced arguments to almost full accuracy. Maybe, that in rare cases this does not hold. Nevertheless, in any case, it is correct, that the computed interval $[t]$ contains the value of the reduced argument t, t computed in the field of the real numbers. The inclusion property $\sin[x] \subseteq \sin\diamondsuit([x])$ will not be corrupted. The notation $\sin\diamondsuit([x])$ means the result as it is computed on the computer.

9.5.3.2 Improved Interval Logic for the Sine Function

In many cases it is possible to compute an enclosure of $\sin[x]$ using only one function evaluation $\sin\diamondsuit([t])$. We only have to supplement the argument reduction step appropriately. The additional information can be used, to construct $[\underline{r},\overline{r}] \supseteq \sin[x]$ in an efficient manner.
The reduction step gives back not only the value k, which is computed by

$$u := x \boxdot \left.\frac{2}{\pi}\right|_{\text{IEEE}} \tag{9.24}$$

$$k := \text{round}(u) \tag{9.25}$$

but also the integer value

$$\text{q} := \text{ greatest integer less than or equal to } \left.\frac{2}{\pi}\right|_{\text{IEEE}} . \tag{9.26}$$

The quantity $1 + (\text{q mod } 4)$ indicates the position of the original argument x (interpreted as an angle given in radians) in the plane. Here all angles $u+2k\pi,\ 0 \le u < 2\pi$ lead to the same value of q.
To find an enclosure of $\sin[x]$ we proceed as follows:

1. **if** $\text{diam}[x] \ge 2\pi$ **then return** $r := [-1, 1]$
2. reduce the point argument $\underline{x}$ to get $[\,\underline{t}\,], \underline{k}$ and $\underline{q}$ according to (9.23), (9.24), (9.25) and (9.26)
3. reduce $\overline{x}$ to get $[\,\overline{t}\,], \overline{k}, \overline{q}$ according to (9.23), (9.24), (9.25) and (9.26)
4. use $\underline{k}, \underline{q}$ and $\overline{k}, \overline{q}$ to find the appropriate case according to Figure 9.7 , i. e. find m and M according to Figure 9.7
5. **return**

$$\underline{r} := \sin(m) \text{ and } \overline{r} := \sin(M)$$

Now $\sin([x]) \subseteq [\underline{r}, \overline{r}]$.

Step 4. has to be explained in more detail. Figure 9.7 shows all relavant cases of different positions of $[x]$. Other cases lead to the result $[r] = [-1, 1]$ because $\mathrm{diam}[x] \geq 2\pi$ or they can be reduced to one of the indicated positions by adding a suitable multiple of 2π to $[x]$. Because the sine is a periodic function with period 2π this does not alter the resulting interval.
The argument $[x]$ is shown as a line segment parallel to the x-axes. The positions where sin takes on its maximum or its minimum in $[x]$ are indicated by m or M, respectively. That means $r = [\underline{r}, \overline{r}] = [\sin(m), \sin(M)] = \sin[x]$. If m is an interior point of $[x]$, i.e. $\underline{x} < m < \overline{x}$, then $\sin(m)$ must not be calculated. In this case $\sin(m) = -1$. Similarly, if $\underline{x} < M < \overline{x}$, then $\sin(M) = +1$. Again, no function call is necessary. Figure 9.7 shows, that in many cases only one function call suffices to find $[r] = \sin[x]$.
In the PASCAL-XSC program given below the cases indicated by (*) are not handled separately. In both cases the minimum is computed as $\underline{r} = \min\{\sin \underline{x}, \sin \overline{x}\}$. Similarly, in the cases indicated by (**) $\overline{r} = \max\{\sin \underline{x}, \sin \overline{x}\}$.

9.5.3.3 Algorithmic Description of the Improved Version

The first algorithm summarizes the accurate argument reduction. The second algorithm shows the improved 'interval logic'.

Algorithm 9.7: ArgRed $(x, [t], k, q)$ {subroutine}

1. $u := x \boxdot \left.\frac{\pi}{2}\right|_{\text{IEEE}}$ {floating-point number}
2. $k := \mathrm{round}(u)$ {integer}
3. $q := \mathrm{floor}(u)$ {integer}
4. $[t] := \Diamond(x - \sum_{j=1}^{jMax} kc_j - k[c_{jMax}])$ {interval scalar product results in an interval}
5. **return** $[t], k, q$

Algorithm 9.8: iSin { }

if $\mathrm{diam}[x] = 0$ **then**

ArgRed $(\underline{x}, [\,\underline{t}\,], \underline{q}, \underline{k})$

$[s] := \sin([\,\underline{t}\,], \underline{k} \bmod 4)$

else if $\mathrm{diam}[x] \geq 2\pi$ **then** $[s] := [-1, 1]$
else

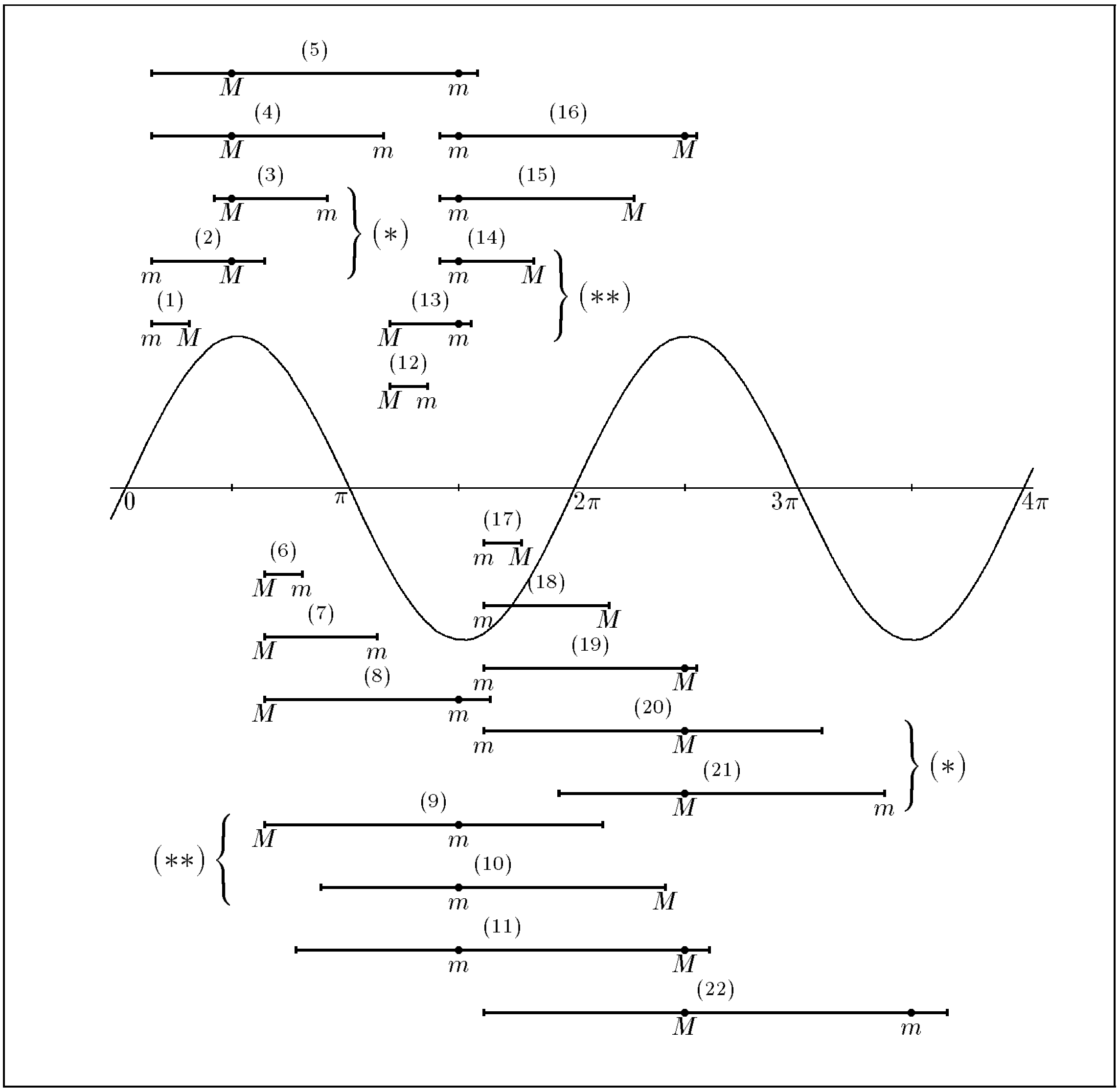


Figure 9.7: Sine Function – Interval Logic

$\mathsf{ArgRed}\,(\underline{x}, [\,\underline{t}\,], \underline{q}, \underline{k})$

$\mathsf{ArgRed}\,(\overline{x}, [\,\overline{t}\,], \overline{q}, \overline{k})$

if $\underline{q} = \overline{q}$ **then**

  if $\mathrm{diam}[x] \geq \pi$ **then** $[s] := [-1, 1]$

  else $[s] := \sin([\,\underline{t}\,], \underline{k}) \,\underline{\cup}\, \sin([\,\overline{t}\,], \overline{k})$

else

  if $\underline{q} = 0$ **then**

    if $\overline{q} = 1$ **then** $[s] := [\min\{\sin([\underline{t}], \underline{k}), \sin([\overline{t}], \overline{k})\}, 1]$
    else if $\overline{q} = 2$ **then** $[s] := [\sin([\overline{t}], \overline{k}), 1]$
    else $[s] := [-1, 1]$

else if $\underline{q} = 1$ **then**
  if $\overline{q} = 0$ **then** $[s] := [-1, \max\{\sin([\underline{t}], \underline{k}), \sin([\overline{t}], \overline{k})\}]$
  else if $\overline{q} = 2$ **then** $[s] := \sin([\underline{t}], \underline{k}) \underline{\cup} \sin([\overline{t}], \overline{k})$
  else $[s] := -1 \underline{\cup} \sin([\overline{t}], \overline{k})$
else if $\underline{q} = 2$ **then**
  if $\overline{q} = 0$ **then** $[s] := -1 \underline{\cup} \sin([\overline{t}], \overline{k})$
  else if $\overline{q} = 1$ **then** $[s] := [-1, 1]$
  else $[s] := [-1, \max\{\sin([\underline{t}], \underline{k}), \sin([\overline{t}], \overline{k})\}]$
else if $\underline{q} = 3$ **then**
  if $\overline{q} = 0$ **then** $[s] := \sin([\underline{t}], \underline{k}) \underline{\cup} \sin([\overline{t}], \overline{k})$
  else if $\overline{q} = 1$ **then** $[s] := \sin([\underline{t}], \underline{k}) \underline{\cup} 1$
  else $[s] := [\min\{\sin([\underline{t}], \underline{k}), \sin([\overline{t}], \overline{k})\}, 1]$

1. **return** iSin $:= [s]$

Here the minimum (maximum) of two intervals means the minimum (maximum) of the two lower (upper) bounds of these intervals. The symbol $\underline{\cup}$ denotes the interval hull of its two arguments.

9.5.3.4 Program Code

We now give the PASCAL-XSC code of the module `sinmi` containing the improved implementation `iSin`.

```
MODULE sinmi;    { Sine Module Improved Versuion }

{ Enclosure of sin([x])  (improved version) }

USE i_ari;    { Make available interval arithmetic             }
USE fnc_util; { Make available Factorial, Min, Max, Floor, ... }
USE mp_ari;   { Make available multi-precision arithmetic      }

CONST SinEps = 1e-15; { Bound for the approximation error of sin }
CONST CosEps = 1e-15; { Bound for the approximation error of cos }

CONST SinMax= 7;       { Degree of the polynomial approximation for sin }
CONST CosMax= 8;       { Degree of the polynomial approximation for cos }
VAR a: ARRAY[0..SinMax] OF interval;  { polynomial coefficients for sin }
VAR b: ARRAY[0..CosMax] OF interval;  { polynomial coefficients for cos }

{ Variables for Sin and Cos to be initialized in the body of this module }
VAR
   SinUpDown: interval;
   CosUpDown: interval;

FUNCTION Sin(t: interval): interval;
{ Sine function for interval arguments t in the range [-pi/4, pi/4] }
{ Notice: The intrinsic sin of PASCAL-XSC is overwritten in this module }
VAR
  k: integer;
  s, tt: interval;
  { Global variables are SinMax and polynomial coefficients a[0..SinMax] }
  {                      and SinUpDown;     t in [-pi/4, pi/4 ]          }
BEGIN
```

```
  { Perform interval Horner scheme for sin }
  tt:= t*t;
  s:= a[SinMax];
  FOR k:= SinMax-1 DOWNTO 0 DO BEGIN
    s:= s*tt + a[k];
  END;
  s:= s*t;
  { Take approximation error into account }
  Sin:= s*SinUpDown;
END;

FUNCTION Cos(t: interval): interval;
{ Cosine function for interval arguments t in the range [-pi/4, pi/4]   }
{ Notice: The intrinsic cos of PASCAL-XSC is overwritten in this module }
VAR
  k: integer;
  s, tt: interval;
  { Global variables are CosMax and polynomial coefficients b[0..CosMax] }
  {                      and CosUpDown                                   }
BEGIN
  { Perform interval Horner scheme for cos }
  tt:= t*t;
  s:= b[CosMax];
  FOR k:= CosMax-1 DOWNTO 0 DO BEGIN
    s:= s*tt + b[k];
  END;
  { Take approximation error into account }
  Cos:= s*CosUpDown;
END;


FUNCTION Sin(r: interval; n: integer): interval;
{ sin(x) with x reduced to r in [-pi/4, pi/4]                      }
{   r: reduced argument                                            }
{   n: indicates whether the original argument x is in the range }
{        [-pi/4, pi/4  ]   + k*2*pi ==>  n = 0                     }
{        [ pi/4, 3*pi/4]   + k*2*pi ==>  n = 1                     }
{        [ 3*pi/4, 5*pi/4] + k*2*pi ==>  n = 2                     }
{        [ 5*pi/4, 7*pi/4] + k*2*pi ==>  n = 3                     }
{  n = round(x*2/pi) mod 4 is computed along with the argument reduction }

VAR sinx: interval;
{ Calls to Sin and Cos are calls to the corresponding functions }
{   implemented in this module                                  }
BEGIN
  IF n = 0 THEN BEGIN
    sinx:= Sin(r);
  END
  ELSE IF n = 1 THEN BEGIN
    sinx:= Cos(r);
  END
  ELSE IF n = 2 THEN BEGIN
    sinx:= -Sin(r);
  END
  ELSE  BEGIN { n = 3 }
    sinx:= -Cos(r);
  END;
  Sin:= sinx;
END;

{ Variables to be initialized for iSin in the body of this module }
VAR
   One: interval;
   TwoOverPi: real;
```

```
   PiOverTwo: interval;  { Enclosure of Pi/2 to IEEE precision }
   PiOverTwoHigh  : real;       { Higher                       }
   PiOverTwoMiddle: real;       {    precision                 }
   PiOverTwoLow   : interval;   {         enclosure of Pi/2    }
   MaxInt   : integer;
   SinHuge  : real;


{ Global variables are PiOverTwo, TwoOverPi, SinHuge               }

PROCEDURE Reduction(x: real; VAR RedArg: interval; VAR Quad, n: integer);
VAR
   k: integer;
   kReal: real;
   r: interval;
BEGIN
  kReal:= x*TwoOverPi;
  k:= round(kReal);                       { round to nearest integer }
  r:= ##(x - k*PiOverTwoHigh - k*PiOverTwoMiddle - k*PiOverTwoLow);
  n:= k MOD 4; { see description of function rsin }
  RedArg:= r;
  Quad:= Floor(kReal) MOD 4;  { x mod 2*Pi lies in quadrant Quad+1 }
END;

{-------------------------------------------------------------------}
{ Purpose: Computation of an enclosure of the range of the sine      }
{          function over the interval x.                             }
{ Parameters:                                                        }
{    In     : 'x'     : interval over which the sine function is to be }
{                          evaluated                                 }
{    Out    : 'iSin'  : range of values (function name)              }
{-------------------------------------------------------------------}
{                                                                    }
{ We do  n o t  overwrite the intrinsic sin globally. Instead we     }
{   define a new function iSin. This allows calls to the intrinsic   }
{   sine as well as calls to our new routine iSin outside this module }
{                                                                    }
{-------------------------------------------------------------------}

GLOBAL FUNCTION iSin(x: interval): interval;
VAR
  rlb: interval;  { reduced argument corresponding to inf(x) }
  rub: interval;  { reduced argument corresponding to sup(x) }
  qlb, nlb: integer; { additional information corresp. to rlb }
  qub, nub: integer; { additional information corresp. to rub }
  PiDown: real;
  TwoPiDown: real;
BEGIN
  IF (abs(inf(x)) > SinHuge) OR (abs(sup(x)) > SinHuge) THEN BEGIN
    writeln(' Argument in iSin out of range!!! ');
    rlb:= 1/0.0;
  END;
  PiDown:= pred(4*arctan(1.0));
  TwoPiDown:= pred(8*arctan(1.0));
  IF diam(x) = 0 THEN BEGIN
    Reduction(x.inf, rlb, qlb, nlb);
    iSin:= Sin(rlb, nlb);
  END
  ELSE BEGIN
    IF ( x.sup -< x.inf ) > TwoPiDown THEN BEGIN
      iSin:= intval(-1, 1)
    END
    ELSE BEGIN
      Reduction(x.inf, rlb, qlb, nlb);
```

```
      Reduction(x.sup, rub, qub, nub);
      writeln('qlb,    qub : ', qlb, '   ',qub);
      writeln('nlb,    nub : ', nlb, '   ',nub);
      IF qlb = qub THEN BEGIN
        IF x.sup -> x.inf > PiDown THEN
          iSin:= intval(-1, 1)
        ELSE IF (qlb = 1) OR (qlb = 2) THEN
          iSin:= Sin(rub, nub) +* Sin(rlb, nlb)
        ELSE
          iSin:= Sin(rlb, nlb) +* Sin(rub, nub)
      END
      ELSE { qlb <> qub } BEGIN
        IF qlb = 0 THEN BEGIN
          IF qub = 1 THEN
            iSin:= intval(min(Sin(rlb, nlb), Sin(rub, nub)), 1)
          ELSE IF qub = 2 THEN
            iSin:= Sin(rub, nub) +* One
          ELSE { if qub = 3 }
            iSin:= intval(-1, 1)
        END
        ELSE IF qlb = 1 THEN BEGIN
          IF qub = 0 THEN
            iSin:= intval(-1, max(Sin(rlb, nlb), Sin(rub, nub)))
          ELSE IF qub = 2 THEN
            iSin:= Sin(rub, nub) +* Sin(rlb, nlb)
          ELSE { if qub = 3 }
            iSin:= intval( -1, sup(Sin(rlb, nlb)) )
        END
        ELSE IF qlb = 2 THEN BEGIN
          IF qub = 0 THEN
            iSin:= intval( -1, sup(Sin(rub, nub)) )
          ELSE IF qub = 1 THEN
            iSin:= intval(-1, 1)
          ELSE { if qub = 3 }
            iSin:= intval(-1, max(Sin(rlb, nlb), Sin(rub, nub)))
        END
        ELSE IF qlb = 3 THEN BEGIN
          IF qub = 0 THEN
            iSin:= Sin(rlb, nlb) +* Sin(rub, nub)
          ELSE IF qub = 1 THEN
            iSin:= Sin(rlb, nlb) +* One
          ELSE { if qub = 2 }
            iSin:= intval(min(Sin(rlb, nlb), Sin(rub, nub)), 1)
        END;
      END;
    END;
  END;
END;

{ Variables only used in the initialization part }
VAR k, sign: integer;
    MpPiOverTwo: mpreal; { multi-precision variable }
    h: real;

BEGIN  { Initialization part }
   SinUpDown:= intval( 1.0 -< SinEps, 1.0 +> SinEps );
   CosUpDown:= intval( 1.0 -< CosEps, 1.0 +> CosEps );
   a[0]:= 1;
   sign:= -1;
   FOR k:= 1 TO SinMax DO BEGIN
      a[k]:= sign*intval(1)/Factorial(2*k+1); { approximation for sin }
      sign:= -sign;
   END;
   b[0]:= 1;
```

```
    sign:= -1;
    FOR k:= 1 TO CosMax DO BEGIN
       b[k]:= sign*intval(1)/Factorial(2*k);   { approximation for cos }
       sign:= -sign;
    END;
    PiOverTwo:= 2*arctan(intval(1.0)); { Enclosure of 2/Pi }
    TwoOverPi:= 2/(4*arctan(1.0));
    MaxInt:= 2147483647;                       { 2**31 - 1 }
    SinHuge:= MaxInt*inf(PiOverTwo);

    One:= 1;
    setprec(6);
    mpinit(MpPiOverTwo);
    MpPiOverTwo:= 1.0;  { Conversion two multi-precision data type }
    MpPiOverTwo:= 2*arctan(MpPiOverTwo);         { = 2*Pi/4 = Pi/2 }
    PiOverTwoHigh   := MpPiOverTwo;
    PiOverTwoMiddle:= MpPiOverTwo - PiOverTwoHigh;
    h:= (MpPiOverTwo - PiOverTwoHigh) - PiOverTwoMiddle;
    PiOverTwoLow    := intval(pred(h), succ(h));

    { Now it holds:  Pi/2 is an element of              }
    {   PiOverTwoHigh + PiOverTwoMiddle +  PiOverTwoLow  }
 END.
```

9.5.3.5 Numerical Examples using the Improved Sine Routine

The following program `sinmitst` computes enclosures for sin([r,r]), r varying in the neighbourhood of $10000 \cdot \pi$. The intrinsic PASCAL-XSC function `pred` computes the predecessor and the function `succ` the successor of a floating-point number.

```
{-----------------------------------------------------------------}
{ Driver for the module sinmi (Improved Version of Sine)          }
{-----------------------------------------------------------------}
PROGRAM sinmitst;

USE i_ari;     { Interval operations                             }
USE sinmi;     { Module with improved version for interval sine }
USE fnc_util; { Make available functions NextAfter, Max, ...    }

VAR
   i, factor: integer;
   x, r: real;
BEGIN
   factor:= 10000;
   x:=4*arctan(1.0);                       { approximation to pi }
   x:= factor*x;
   writeln(
     'Check the accuracy of the improved version iSin(x) around ',
      factor:0, '*Pi:');
   r:= NextAfter(x, -5); { go left 5 flp-numbers }
   FOR i:= 1 TO 10 DO BEGIN
     r:= succ(r);
     writeln( iSin(intval(r,r)) );
   END;
   r:= power(2,30);
   writeln; writeln('Compute iSin(2**30) = iSin(', r:12:1, '):');
   writeln( iSin(intval(r,r)) );
END.
{-----------------------------------------------------------------}
```

The output of the driver program is as follows:

```
Check the accuracy of the improved version iSin(x) around 10000*Pi:
[  -1.50375975823238E-011,  -1.50375975823236E-011 ]
[  -1.13996187752321E-011,  -1.13996187752319E-011 ]
[  -7.76163996814029E-012,  -7.76163996814026E-012 ]
[  -4.12366116104857E-012,  -4.12366116104855E-012 ]
[  -4.85682353956850E-013,  -4.85682353956848E-013 ]
[   3.15229645313485E-012,   3.15229645313487E-012 ]
[   6.79027526022656E-012,   6.79027526022659E-012 ]
[   1.04282540673182E-011,   1.04282540673184E-011 ]
[   1.40662328744099E-011,   1.40662328744101E-011 ]
[   1.77042116815016E-011,   1.77042116815018E-011 ]

Compute iSin(2**30) = iSin(1073741824.0):
[  -6.17326415046044E-001,  -6.17326415046041E-001 ]
```

Compare the width of the computed enclosure of $\sin(2^{30})$ with the width of the enclosure as it was computed in Subsection 9.5.2.3 using the more simple algorithm `SinSimp`.

9.6 Exponential Function

In this section we start with a simple method to compute enclosures for $\exp(x)$. An improved version using an a priori error estimation will be discussed in Subsection 9.6.2.2.

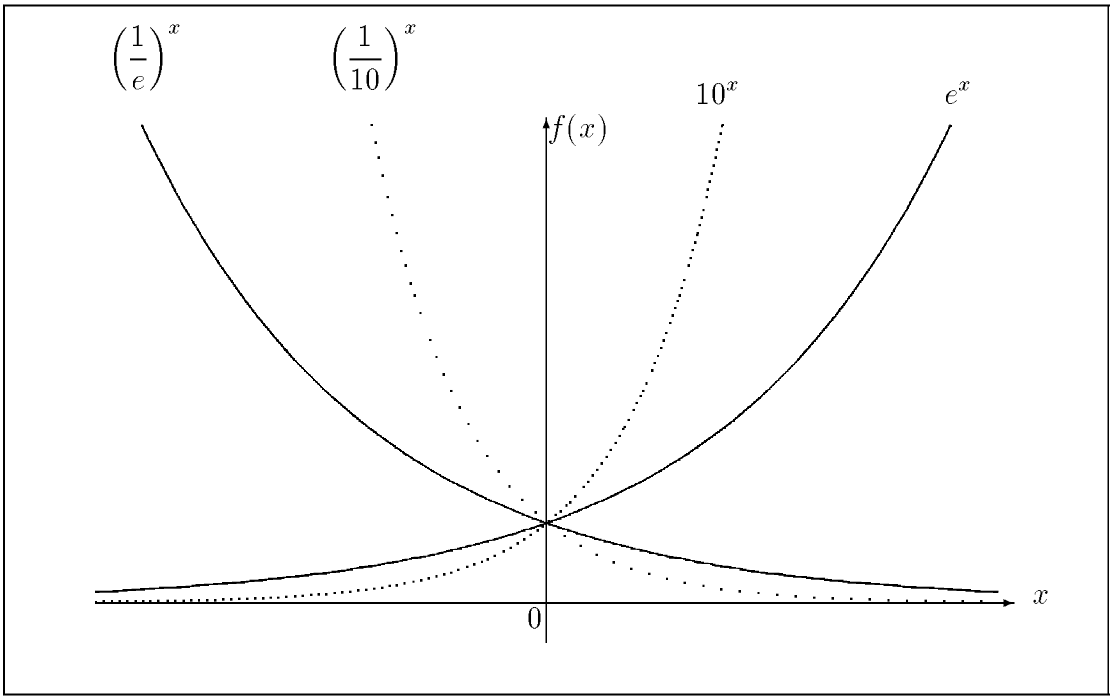


Figure 9.8: Exponential Function

9.6.1 Exponential Function for Point Arguments

The computation of the exponential $\exp(x) = e^x$ is done in three steps:

1. argument reduction into a basic interval
2. approximation within the basic interval
3. result adaptation.

9.6.1.1 Argument Reduction

Because

$$\exp(x) = \begin{cases} \exp(|x|) & , \quad x \geq 0 \\ \dfrac{1}{\exp(|x|)} & , \quad x < 0 \end{cases} \tag{9.27}$$

we restrict the range reduction without loss of generality to positive arguments $x \geq 0$.
Then we may write

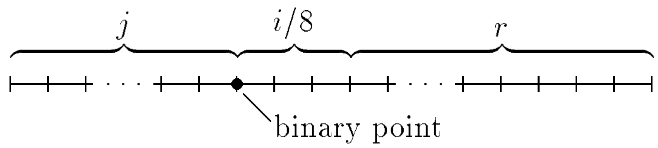
$$e^x = 2^{\log_2 e^x} = 2^{x \cdot \log_2 e} = 2^{x \cdot \frac{1}{\ln 2}} =: 2^{j + i/8 + r} \tag{9.28}$$

with j integer, $i \in \{0, 1, \ldots, 7\}$, and $0 \leq r < \dfrac{1}{8}$.
The value

$$e^x = 2^{j + i/8 + r} = 2^j \cdot 2^{i/8} \cdot 2^r \tag{9.29}$$

may be computed by an approximation to 2^r, $0 \leq r < 1/8$, one additional multiplication by the prestored constant $2^{i/8}$ and by adding j to the exponent of the resulting product.
The value j is nothing else but the integral part of $h := x/\ln 2$ and i is given by the bit combination of the first three bits behind the binary point of h.

$h = \dfrac{x}{\ln 2}$



Range Reduction: Splitting of h in three parts

The quantities j, i and r are computed as follows:

$$\left.\begin{array}{rcll} h & := & x \cdot \dfrac{1}{\ln 2} & \\ j & := & \mathrm{trunc}(h) & \\ h & := & h - j & \\ i & := & \mathrm{trunc}(8 \cdot h) & \in \{0, 1, \ldots, 7\} \\ r & := & h - \dfrac{i}{8} & \end{array}\right\} \tag{9.30}$$

Thus $\dfrac{x}{\ln 2} = j + \dfrac{i}{8} + r$ with $r \in [0, \dfrac{1}{8})$.

9.6.1.2 Approximation Within the Basic Interval

We are interested in the value of 2^r for arguments $r \in [0, \frac{1}{8})$.

$$2^r = e^{r \cdot \ln 2} = \sum_{k=0}^{\infty} \frac{(\ln 2)^k}{k!} r^k = \sum_{k=0}^{\infty} q_k r^k \tag{9.31}$$

with coefficients

$$q_k := \frac{(\ln 2)^k}{k!}. \tag{9.32}$$

Truncating this series after $n+1$ summands leads to the relative approximation error

$$\begin{aligned} \left| \frac{2^r - \sum\limits_{k=0}^{n} q_k r^k}{2^r} \right| &= \left| \frac{\sum\limits_{k=n+1}^{\infty} q_k r^k}{2^r} \right| \\ &\leq q_{n+1} r^{n+1} \sum_{k=n+1}^{\infty} \underbrace{\frac{q_k}{q_{n+1}}}_{\in[0,1]} r^{k-(n+1)} \\ &\underset{(0 \leq r < 1/8)}{\leq} q_{n+1} r^{n+1} \cdot \frac{1}{1-r} \end{aligned} \tag{9.33}$$

Using $n = 9$ we find as an upper bound for the relative approximation error

$$q_{10} r^{10} \frac{1}{1-r} \underset{(0 \leq r < 1/8)}{\leq} \frac{(\ln 2)^{10}}{10!} \left(\frac{1}{8}\right)^{10} \underbrace{\frac{1}{1-\frac{1}{8}}}_{<1.15} < 7.6 \cdot 10^{-18}. \tag{9.34}$$

So the approximation

$$p_9(r) := \sum_{k=0}^{10} \frac{(\ln 2)^k}{k!} r^k \approx 2^r, \qquad r \in [0, \frac{1}{8}] \tag{9.35}$$

is appropriate for the IEEE 64 bit floating-point screen (see (9.5)) .

9.6.1.3 Result Adaptation

As result adaptation we have to compute

$$e^x = 2^{\frac{\ln 2}{x}} = 2^j \cdot 2^{i/8} \cdot 2^r, \tag{9.36}$$

j, i and r according to (9.30). The integer i ranges from 0 to 7. Therefore we must know the constants

$$c_i := 2^{i/8} = \sqrt[8]{2^i}, \quad i = 0, 1, \ldots, 7. \tag{9.37}$$

These quantities are stored as precomputed IEEE floating-point numbers in a small table. In order to get an enclosure of the function value $\exp(x)$ actually the constants are enclosed by floating-point intervals.

9.6.1.4 Algorithms

The Argument reduction for $\exp(x)$ is done by

Algorithm 9.9: ArgRed $(x, [r], j, i)$ {subroutine}

1. $[h] := x \cdot \Diamond(\frac{1}{\ln 2})$ $\qquad \{[h] = [\underline{h}, \overline{h}]\}$
2. $j := \mathrm{trunc}(\underline{h})$
3. $[h] := [h] - j$
4. $i := \mathrm{trunc}(8 \cdot \underline{h})$
5. $[h] := [h] - i/8$
6. **return** $[r] := [h], j, i$

The approximation within the basic range is as follows:

Algorithm 9.10: Approx $([r])$ {function}

{It is assumed $[q_k] \ni \dfrac{(\ln 2)^k}{k!} = q_k, \quad k = 0, 1, \ldots, n$}

$[s] := [q_n]$
for $k := n-1$ **downto** 0 **do**
  $[s] := [s] \cdot [r] + [q_k]$
$[s] := [s] \cdot [1-\varepsilon, 1+\varepsilon]$ $\qquad$ {ε bounds the approximation error}
$\qquad$ {it holds $2^{[r]} \subseteq [s]$}
return Approx $:= [s]$

An enclosure for $\exp(x)$ for a point argument x may be computed by

Algorithm 9.11: ExpSimp (x) {function}

{It is assumed: $2^{i/8} \in [c_i], i = 0, 1, \ldots, 7$}

if $x > MaxExpArg$ **then**

$ERROR$

else if $x = 0$ **then** $[s] := 1$

else if $x < MinArg$ **then** $[s] := [0, succ(0.0)]$

else

$neg := (x < 0)$

ArgRed $(abs, [r], j, i)$

$[s] :=$ Approx $([r])$ {it holds $2^{[r]} \subseteq [s]$ }

$[s] := 2^j \cdot [s] \cdot [c_i]$ {it holds $e^{|x|} \in [s]$ }

if neg **then** $[s] := 1/[s]$

{now it holds $e^x \in [s]$ }

return ExpSimp $:= [s]$

9.6.1.5 Program Code

In the following PASCAL-XSC module `expm` Algorithm 9.10 (Approx $([r])$) is not implemented as a separate function. Instead, it is included in the main routine ExpSimp. We now give the PASCAL-XSC source code for the module `expm`. Here the simple program version `ExpSimp` which computes an interval enclosure of $\exp(x)$ is implemented.

```
{----------------------------------------------------------------------}
{ Module expm:  Exponential Module Simple Version                      }
{----------------------------------------------------------------------}
MODULE expm;

USE i_ari;     { Interval operations                     }
USE fnc_util;  { Make available function Pow2(n) = 2**n }

{----------------------------------------------------------------------}
{ Variables that are initialized in the body of this module for the }
{ function   ExpSimp(x: real): interval                                }
{----------------------------------------------------------------------}
VAR
  {                                                                    }
  { polynomial coefficients  q   :=  ln(2)**k / k!                     }
  {                           k                                        }

  q: ARRAY[0..9] OF interval;

  RedConst: interval;   { reduction constant 1/ln(2)                   }

  { Constants for the result adaptation:                               }
  {                                                                    }
  {    ResConst[k]:= 2**(k/8) , k = 0, 1, ..., 7                       }
```

```
  ResConst: ARRAY[0..7] OF interval;

  MaxExpArgument: real; { overflow bound  }
  MinArgument   : real; { underflow range }
  MinRealNormalized: real; { smallest pos. normalized flp-number    }
  Eps              : real; { bound for the approximation error      }
  OneDownUp        : interval; { [1 - Eps, 1 + Eps]                 }

{-------------------------------------------------------------------}
{ Exponential for point arguments                                   }
{-------------------------------------------------------------------}
GLOBAL FUNCTION ExpSimp(x: real): interval;
VAR
  s      : interval;  { result                  }
  r      : interval;  { auxiliary variable      }
  t      : interval;  { reduced argument        }
  neg    : boolean;   { indicates whether x is negative }
  i      : integer;   { index for result adaptation     }
  j      : integer;   { integral part of x/ln(2)        }
  k      : integer;   { loop variable                   }
BEGIN
  IF (x > MaxExpArgument) THEN BEGIN
    writeln('*** Argument too large in routine ExpSimp ***');
    x:= x/0;  { force termination of the program }
  END
  ELSE IF (x = 0) THEN
    ExpSimp:= 1
  ELSE IF (x < MinArgument) THEN
    ExpSimp:= intval(0.0, MinRealNormalized)
  ELSE BEGIN
    neg:= x < 0;
    IF neg THEN x:= -x;

    { Perform argument reduction }

    r:= x*RedConst;             { r = x/ln(2) }

    j:= trunc(inf(r));          { integral part of x/ln(2)           }

    r:= r - j;            { remove bits in front of binary point    }

    i:= trunc(8*inf(r));  { index for result adaptation             }

    r:= r - i*0.125;      { remove first 3 bits behind binary point }

    t:= r;                { reduced argument                        }

    { Horner's scheme using interval arithmetic: }

    { Compute an approximation to 2**t       }

    s:= q[9];
    FOR k:= 8 DOWNTO 0 DO
      s:= s*t + q[k];

    { Now: s is close to 2**t }

    { Result adaptation: }

    IF neg THEN BEGIN

      s:= Pow2(-j)/(s*ResConst[i]);

      { Take approximation error into account: }
```

```
      IF s = 0 THEN
        ExpSimp:= intval(0, MinRealNormalized)
      ELSE
        ExpSimp:= s*OneDownUp;
    END
    ELSE BEGIN

      s:= Pow2(j)*s*ResConst[i];

      { Take approximation error into account: }
      ExpSimp:= s*OneDownUp;
    END;
  END;
END;

{----------------------------------------------------------------------}
{ Exponential for interval arguments                                   }
{----------------------------------------------------------------------}
GLOBAL FUNCTION ExpSimp(x: interval): interval;
VAR
  infx, supx: real;
BEGIN
  infx:= inf(x);
  supx:= sup(x);
  IF infx = supx THEN
    { argument is a point interval }
    ExpSimp:= ExpSimp(infx)
  ELSE
    { exp(x) is monotone increasing }
    ExpSimp:= intval( inf(ExpSimp(infx)), sup(ExpSimp(supx)) );
END;


{----------------------------------------------------------------------}
VAR
  ln2, p, f: interval;  { Local to the initialization part }
  k: integer;           { Loop variable                    }

BEGIN
  {--------------------------------------------------------------}
  { Initialization part                                          }
  {--------------------------------------------------------------}
  MaxExpArgument    := 709.782712893384;
  MinArgument       := -MaxExpArgument;
  MinRealNormalized := 2.2250738585072013e-308;
  Eps               := 4.0E-16;
  OneDownUp         := intval( 1 -< Eps, 1 +> Eps );

  {--------------------------------------------------------------}
  { Enclosure of ln(2) (see Hart, page 182):                     }
  {--------------------------------------------------------------}

  ln2:= intval(0.6931471805599453094172321,
                0.6931471805599453094172322 );

  {--------------------------------------------------------------}
  { Compute enclosures for the polynomial coefficients           }
  {                                                              }
  {              ln(2)**k                                        }
  {  q[k]    :=  -------- , k = 0, 1, ..., 9                     }
  {                k!                                            }
  {                                                              }
  {--------------------------------------------------------------}
```

```
  f:= 1;
  p:= 1;
  q[0]:= 1;
  FOR k:= 1 TO 9 DO BEGIN
    p:= p*ln2;               { ln(2)**k }
    f:= f*k;                 { k!       }
    q[k]:= p/f;
  END;

  {-------------------------------------------------------------}
  { Reduction constant: enclosure of 1/ln(2)                    }
  {-------------------------------------------------------------}
  RedConst:= 1/ln2;
  {   1/ln(2) =    1.44269504088896340735992468100189214 }

  {-------------------------------------------------------------}
  { Constants for result adaptation                             }
  {   2**(k*0.125) = sqrt(sqrt(sqrt(2**k))),  k = 0, 1, ..., 7  }
  {-------------------------------------------------------------}
  ResConst[0]:=  1;
  ResConst[1]:=  (sqrt(sqrt(sqrt(intval(  2.0)))));
  ResConst[2]:=  (sqrt(sqrt(sqrt(intval(  4.0)))));
  ResConst[3]:=  (sqrt(sqrt(sqrt(intval(  8.0)))));
  ResConst[4]:=  (sqrt(sqrt(sqrt(intval( 16.0)))));
  ResConst[5]:=  (sqrt(sqrt(sqrt(intval( 32.0)))));
  ResConst[6]:=  (sqrt(sqrt(sqrt(intval( 64.0)))));
  ResConst[7]:=  (sqrt(sqrt(sqrt(intval(128.0)))));

END.                                         { Initialization part }
{---------------------------------------------------------------}
```

9.6.1.6 Numerical Examples

The driver program `expmtst` computes enclosures for some point arguments.

```
PROGRAM expmtst;    { Driver for Exponential Module Simple Version }

USE i_ari,  { Interval operations }
    expm;   { Exponential module simple version }

VAR
  k  : integer;  { Loop variable                                 }
  arg: integer;  { For better readability we use integer arguments }
  res: interval; { Enclosure for exp(arg)                        }
BEGIN
  arg:= -50;
  FOR k:= 1 TO 6 DO BEGIN
     arg:= arg + 50;
     res:= ExpSimp(arg);
     writeln('ExpSimp(', arg:3, ')=', res);
  END;
  writeln('Use the intrinsic PASCAL-XSC exp routine:');
  res:= exp(arg);
  writeln('exp(     ', arg:3, ')=', res);
END.
```

This program `expmtst` produces the following output:

```
ExpSimp(  0)=[  1.000000000000000E+000,  1.000000000000000E+000 ]
ExpSimp( 50)=[    5.1847055285870E+021,    5.1847055285872E+021 ]
ExpSimp(100)=[    2.6881171418161E+043,    2.6881171418163E+043 ]
ExpSimp(150)=[    1.3937095806663E+065,    1.3937095806665E+065 ]
ExpSimp(200)=[     7.225973768125E+086,     7.225973768127E+086 ]
ExpSimp(250)=[     3.746454614502E+108,     3.746454614503E+108 ]
Use the intrinsic PASCAL-XSC exp routine:
exp(    250)=[  3.746454614502673E+108,  3.746454614502674E+108 ]
```

For larger arguments the enclosures become worse. This is due to cancellation effects when reducing the argument.

9.6.2 Improved Version of the Exponential

We now discuss a version for the computation of the exponential which exploits a priory error estimations and which performs an accurate range reduction.

9.6.2.1 Accurate Range Reduction

In 9.6.1.1 we have already discussed the range reduction for the exponential. The critical point is the cancellation of leading bits when substracting the integral part j and the first three bits (given by the value $i/8$) after the binary point of $x/\ln 2$ from $x/\ln 2$. On the machine the value $x/\ln 2$ is computed by $x \boxdot \left.\frac{1}{\ln 2}\right|_{\text{IEEE}}$. Because $1/\ln 2$ is an irrational number it cannot be stored as a floating-point number with finite precision. Thus the leading product in the formula

$$r = x \cdot \frac{1}{\ln 2} - j - i$$

for the reduced argument r is in general not exactly representable. By cancellation of the leading bits (this is the intention of the argument reduction) the error inherent in the machine product is amplified significantly. The reduced argument r loses the number of significand bits needed to express j and i.
To improve the reduction process we use the accurate dot product of PASCAL-XSC. We express the product $x \cdot \frac{1}{\ln 2}$ by

$$x \cdot \frac{1}{\ln 2} = x(c_1 + c_2) = xc_1 + xc_2.$$

Here $c_1 + c_2 \approx \frac{1}{\ln 2}$ to an accuracy of ca. 106 bits (two double numbers). Using such a representation, the reduced argument r is computed by one accurate scalar product and only one final rounding:

$$r = \square(xc_1 + xc_2 - i - j) \tag{9.38}$$

Because the exponential is a rapidly increasing function the argument range for the computer analogon is limited by the relative small upper bound

$$expmax := \ln(maxreal) \quad = \quad 709.782712893384$$

Therefore maximal $\log_2(710) + 3 < 13$ bits are enough to express the sum $j + i/8$. This shows, that r computed by (9.38) is a floating-point number of full accuracy. Using a staggered representation of $1/\ln 2$ with two floating-point summands suffices to get an accurately reduced argument.

9.6.2.2 Improvement by A Priori Error Estimations

To improve our algorithm in speed we do no longer perform an interval horner scheme for the evaluation of the approximation

$$2^r \approx \sum_{k=0}^{9} q_k r^k.$$

The rounding errors and the errors introduced by the fact, that the polynomial coefficients are in general not exactly representable as floating-point numbers are catched by an a priori error estimation. We can use interval software to find an upper bound for these kinds of error. The algorithm used is a simplified version of a method described in [188]. The theory is discussed in Appendix C. A PASCAL-XSC program called `experr` for automaticly doing an a priori error estimation for the exponential function is also given there.
The actual input data for the program `experr` shown in Appendix C are enclosures $[q_k]$ for the exact polynomial coefficients $q_k = \dfrac{(\ln 2)^k}{k!}$ as well as an enclosure of the basic range $[0, \frac{1}{8}]$ covering the reduced argument r of our exponential routine. The output of the program are two positive numbers called α and β with

$$\begin{aligned}&\left|\sum q_k r^k - \text{Machine Horner}(\tilde{q}_k, \tilde{r}, k = 0, 1, \ldots, 9)\right| \\ &\le |p_n(r) - p_n(\tilde{r})| \cdot \alpha \cdot \varepsilon + |p_n(\tilde{r}) - \tilde{p}_n(\tilde{r})| \cdot \beta \cdot \varepsilon(r)\end{aligned} \tag{9.39}$$

for all $r \in [0, \frac{1}{8}]$. A quantity x marked by $\tilde{x}$ indicates the floating-point analogon $\tilde{x}$ to the corresponding real quantity x. The quantities α, β are “worst case” quantities. The numerical values comming from Appendix C are

```
Alpha:   0.811837
Beta :   2.428265
```

They are independent of the actual value of r.
Due to the accurate argument reduction the relative error $\varepsilon(r) := \left|\dfrac{r - \tilde{r}}{r}\right|$ introduced by the reduction process is bounded by $\varepsilon(r) = \varepsilon = 1.01 \cdot 2^{-53}$. Putting together the derived error bounds and the error bound (9.39) for the approximation error, we find

$$\left|\frac{2^r - \tilde{p}_n(\tilde{r})}{2^r}\right| \underset{(2^r \ge 1)}{\le} (\alpha + \beta) \cdot \varepsilon + \varepsilon(\text{app}) =: \varepsilon_1 \tag{9.40}$$

To find an over–all error bound for

$$\left|\frac{e^x - \widetilde{\exp}(x)}{e^x}\right|$$

additionally the error coming from the reconstruction given by (see (9.36))

$$(\tilde{p}_n(\tilde{r}) \cdot 2^j) \odot \widetilde{\sqrt[8]{2^i}}$$

has to be considered. The constants $\tilde{c}_i := \widetilde{\sqrt[8]{2^i}}, i = 0, 1, \ldots, 7$ are of maximum accuracy. Thus we get

$$\begin{aligned}
(\tilde{p}_n(\tilde{r}) \cdot 2^j) \odot \widetilde{\sqrt[8]{2^i}} &= p_n(r) \cdot (1+\varepsilon_1) \cdot 2^j \cdot \sqrt[8]{2^i} \cdot (1+\varepsilon)^2 \\
&= p_n(r) \cdot 2^j \cdot \sqrt[8]{2^i} \cdot (1 + \underbrace{\varepsilon_1 + 2.01\varepsilon}_{=:\,\varepsilon_2}) \\
&= e^x \cdot (1+\varepsilon_2) \qquad (9.41)
\end{aligned}$$

The estimation given so far holds for arguments $x \geq 0$. If $x < 0$ a final division which may be accompanied by an additional rounding error has to be performed. In this case

$$\begin{aligned}
e^x &= \frac{1}{(p_n(r) \cdot 2^j \cdot \sqrt[8]{2^i})(1+\varepsilon_2)}(1+\varepsilon) \\
&= \frac{1}{e^{|x|}}(1 + 1.01(\varepsilon_2 + \varepsilon)) \\
&= e^x(1+\varepsilon_3) \qquad (9.42)
\end{aligned}$$

Summarizing all error estimations leads to

$$\left|\frac{e^x - \widetilde{\exp}(x)}{e^x}\right| = \left|\frac{e^x - e^x(1+\varepsilon_3)}{e^x}\right| \leq \varepsilon(\exp) := 5 \cdot 2^{-53}$$

i.e.

$$e^x \in \widetilde{\exp}(x) \cdot [1 - \varepsilon(\exp), 1 + \varepsilon(\exp)]$$

This representation is used to enclose function values e^x for point arguments x. A special handling is done for $x = 0$ and for arguments $x < \ln(\text{minreal}) := -709.782\ldots$. In the first case the point result $e^0 = [1, 1]$ is returned. In the second case the interval $[0, \text{minreal}]$ is used as an enclosure of e^x.

9.6.3 Exponential Function for Interval Arguments

The exponential function is always positive and monotone increasing. To enclose $\exp[\underline{x}, \overline{x}]$ by a floating-point interval we first compute enclosures explb $\ni e^{\underline{x}}$ and expub $\ni e^{\overline{x}}$ using the routine discussed in the preceding paragraph. Then it holds $\exp[\underline{x}, \overline{x}] \subseteq [\inf(\text{explb}), \sup(\text{expub})]$.

9.6.3.1 Algorithms

First the algorithm for the accurate argument reduction is given.

Algorithm 9.12: AccArgRed $(x, [r], j, i)$ {subroutine}

{c_1, c_2 are two floating-point numbers, the sum of which approximates $1/\ln 2$ to ca. 106 bits}

1. $accu := x \cdot c_1 + x \cdot c_2$ { accurate scalar product }
2. $h := \Box accu$ { *accu* is a *dotprecision* variable }
3. $j := \text{trunc}(h)$
4. $accu := accu - j$
5. $h := \Box(accu)$
6. $i := \text{trunc } 8 \cdot h$
7. $h := \Box(accu - 0.125 \cdot i)$
8. **return** $r := h, j, i$

The next algorithm shows the approximation of $\exp(x)$ within the basic range $[0, 1/8]$. Due to the known a priori error bound only floating-point computations have to be performed.

Algorithm 9.13: Approx $([r])$ {function}

$s := q_n$ $\{ q_k = \dfrac{(\ln 2)^k}{k!} \}$

for $k := n - 1$ **downto** 0 **do**

$s := s \cdot r + q_k$

return Approx $:= s$

Algorithm iExp can be used to compute an enclosure of $\exp(x)$, x being a point argument.

Algorithm 9.14: iExp (x) {function}

$\{c_i := 2^{i/8}, i = 0, 1, \ldots, 7, \quad \varepsilon(\exp)$ according to (9.37) }

if $x > MaxExpArg$ **then**

$ERROR$

else if $x < MinExpArg$ **then**

$[h] := [0, \text{succ}(0.0)]$

else if $x = 0$ **then**

$[h] := 1$

else

$neg := (x < 0)$

ArgRedAcc $(\text{abs}(x), r, j, i)$

$s :=$ Approx (r)

if neg **then**

$[h] := [1 - \varepsilon(\exp), 1 + \varepsilon(\exp)]/(2^j \cdot s \cdot c_i)$

else

$[h] := [1 - \varepsilon(\exp), 1 + \varepsilon(\exp)] \cdot 2^j \cdot s \cdot c_i$

$\{$now it holds $e^x \in [h]$ $\}$

return iExp $:= [h]$

The final algorithm handles $\exp([x])$ for interval arguments.

Algorithm 9.15: iExp $([x])$ {function}

if $\text{diam}[x] = 0$ **then**

$[h] :=$ iExp $(\inf([x]))$

else

$[a] :=$ iExp $(\inf([x]))$

$[b] :=$ iExp $(\sup([x]))$

$[h] := [\inf([a]), \sup([b])]$ $\{$Now $\exp[x] \subseteq [h]$ $\}$

return iExp $:= [h]$

Notice: The function calls of iExp $(\ldots)$ in the body of Algorithm 9.15 refer to Algorithm 9.14 (the arguments are **point** arguments!).

9.6.3.2 Program Code

We now give the PASCAL-XSC source code for the module `expmi`. It implements the improved exponential version `iExp` which computes an interval enclosure of $\exp([x])$.

```
{----------------------------------------------------------------------}
{ Module expmi:  Exponential Module Improved Version                    }
{----------------------------------------------------------------------}
MODULE expmi;  { Exponential Module Improved Version }

USE i_ari;     { Interval operations           }
USE fnc_util;  { Make available Pow2(n):= 2**n  }

{--------------------------------------------------------------------}
{ Variables that are initialized in the body of this module for     }
{   the overloaded iExp functions.                                   }
{--------------------------------------------------------------------}
VAR
  q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10: real;
  {                                                                  }
  { polynomial coefficients  q  :=  ln2**k / k!                      }
  {                           k                                      }

  ResConst: ARRAY[0..7] OF real;
  { Constants for the result adaptation:                             }
```

```
  {                                                                   }
  {    ResConst[k]:= 2**(k/8)                                         }

  RedHigh : real;                 { 1/ln2 high part }
  RedLow  : real;                 { 1/ln2 low part  }
  max_exp_argument   : real; { overflow bound  }
  min_argument       : real; { underflow range }
  min_real_normalized: real; { smallest pos. normalized flp-number  }
  one_down_eps       : real; { error bounds  1 - eps,               }
  one_up_eps         : real; {               1 + eps                }

{--------------------------------------------------------------------}
{ Initialization of the polynomial coefficients, the reconstruction }
{   constants and other values.                                      }
{ The constants are round to nearest values.                         }
{--------------------------------------------------------------------}
PROCEDURE InitExp;
BEGIN
  max_exp_argument   := 709.782712893384;
  min_argument       := -max_exp_argument;
  min_real_normalized:= 2.2250738585072013e-308;
  one_down_eps       := 1 -< 2e-16;
  one_up_eps         := 1 +> 2e-16;

  { Polynomial coefficients ln2**k / k! }
  q0:=   1.0;
  q1:=   6243314768165359.0 /            9007199254740992.0;
  q2:=   8655072057804175.0 /           36028797018963968.0;
  q3:=   7998985059213504.0 /          144115188075855872.0;
  q4:=   5544473941134967.0 /          576460752303423488.0;
  q5:=   6149018367977265.0 /         4611686018427387904.0;
  q6:=   5682899659966343.0 /        36893488147419103232.0;
  q7:=   9003624861053127.0 /       590295810358705651712.0;
  q8:=   6240837187258404.0 /      4722366482869645213696.0;
  q9:=   7690344356767684.0 /     75557863725914323419136.0;
  q10:=  8528864813485770.0 /   1208925819614629174706176.0;

  { Reduction constants: 1/ln2 high and low part }
  RedHigh:=  6497320848556798.0 /                 4503599627370496.0;
  RedLow:=   6605663993728292.0 / 324518553658426726783156020576256.0;

  { Constants for result adaptation 2**(i*0.125), i = 0, 1, ..., 7: }
  ResConst[0]:=  4503599627370496.0 /   4503599627370496.0; { = 1.0 }
  ResConst[1]:=  4911210218475899.0 /   4503599627370496.0;
  ResConst[2]:=  5355712719992597.0 /   4503599627370496.0;
  ResConst[3]:=  5840446135085607.0 /   4503599627370496.0;
  ResConst[4]:=  6369051672525773.0 /   4503599627370496.0;
  ResConst[5]:=  6945500098633947.0 /   4503599627370496.0;
  ResConst[6]:=  7574121564787629.0 /   4503599627370496.0;
  ResConst[7]:=  8259638134547591.0 /   4503599627370496.0;
END;

{--------------------------------------------------------------------}
{ Enclosure of exp(x), x real.                                       }
{ We do  n o t  redefine the intrinsic name exp.                     }
{--------------------------------------------------------------------}
GLOBAL FUNCTION iExp(x: real): interval;
VAR
  s      : real;  { result                  }
  r1, r2 : real;  { reduction               }
  t      : real;  { reduced argument        }
  neg    : boolean;   { indicates whether x is negative }
  i      : integer;   { index for result adaptation     }
  j      : integer;   { integral part of x/ln2          }
```

```
BEGIN

  IF x > max_exp_argument THEN BEGIN
    writeln('*** Argument too large in routine iexp(real) ***');
    x:= x/0;  { force termination of the program }
  END
  ELSE IF x = 0 THEN
    iexp:= 1
  ELSE IF x < min_argument THEN
    iexp:= intval(0.0, min_real_normalized)
  ELSE BEGIN
    neg:= x < 0;
    IF neg THEN x:= -x;

    { Accurate argument reduction: }

    r1:= x*RedHigh;                                   { r1 + r2  = x/ln2 }
    r2:= #<(x*RedHigh + x*RedLow - r1);

    j:= trunc(r1);          { integral part of x/ln2                 }

    r1:= r1 - j;            { remove bits in front of binary point    }

    i:= trunc(8*r1);        { index for result adaptation            }

    r1:= r1 - i*0.125;      { remove first 3 bits behind binary point }

    t:= r1 + r2;            { accurately reduced argument            }

    { Horner's scheme using real arithmetic: }
    { Compute an approximation to 2**t       }

    s:= ((((((((t*q9 + q8)*t+ q7)*t+ q6)*t+ q5)*t
                        + q4)*t+ q3)*t+ q2)*t+ q1)*t+ q0;

    { Now: s is close to 2**t }

    { Result adaptation: }
    IF neg THEN BEGIN

      s:= Pow2(-j)/(s*ResConst[i]);

      { Take approximation error into account: }
      IF s = 0 THEN
        iexp:= intval(0, min_real_normalized)
      ELSE
        iExp:= intval(s*one_down_eps, s*one_up_eps);
    END
    ELSE BEGIN

      s:= Pow2(j)*s*ResConst[i];

      { Take approximation error into account: }
      iExp:= intval(s*one_down_eps, s*one_up_eps);
    END;
  END;
END;
{-------------------------------------------------------------------}


{-------------------------------------------------------------------}
{ Enclosure of exp(x), x interval.                                  }
{ Again we do   n o t  redefine the intrinsic name exp.             }
```

```
{----------------------------------------------------------------}
GLOBAL FUNCTION iExp(x: interval): interval;
VAR
  infx, supx: real;
BEGIN
  infx:= inf(x);
  supx:= sup(x);
  IF infx = supx THEN
    { argument is a point interval }
    iExp:= iExp(infx)
  ELSE
    { exp(x) is monotone increasing }
    iExp:= intval( inf(iExp(infx)), sup(iExp(supx)) );
END;
{----------------------------------------------------------------}

BEGIN
  InitExp; { Initialization }
END.
{------------------------------------------------------------------}
```

9.6.3.3 Numerical Examples

The driver program **expmitst** which uses the same (point) arguments already used in the numerical examples following the simple program version (see Subsection 9.6.1.6) is as follows:
listingstdfktn/expmitst.lst The following output is produced:

```
iExp(  0)=[  1.000000000000000E+000,  1.000000000000000E+000 ]
iExp( 50)=[  5.184705528587072E+021,  5.184705528587075E+021 ]
iExp(100)=[  2.688117141816134E+043,  2.688117141816136E+043 ]
iExp(150)=[  1.393709580666379E+065,  1.393709580666380E+065 ]
iExp(200)=[  7.225973768125748E+086,  7.225973768125752E+086 ]
iExp(250)=[  3.746454614502671E+108,  3.746454614502674E+108 ]
Use the intrinsic PASCAL-XSC exp routine:
exp( 250)=[  3.746454614502673E+108,  3.746454614502674E+108 ]
```

Now all enclosures are of high quality.

9.7 Logarithm

This section presents two algorithms to bound the range of logarithms. The first algorithm computes $\log_2(x)$. The algorithm discussed in Subsection 9.7.2 is a typical table-driven realization of $\ln(x)$.

9.7.1 Enclosing Logarithms

The natural logarithm $\ln x$ is the inverse of $\exp(x)$:

$$\ln(\exp(x)) = \exp(\ln(x)) = x.$$

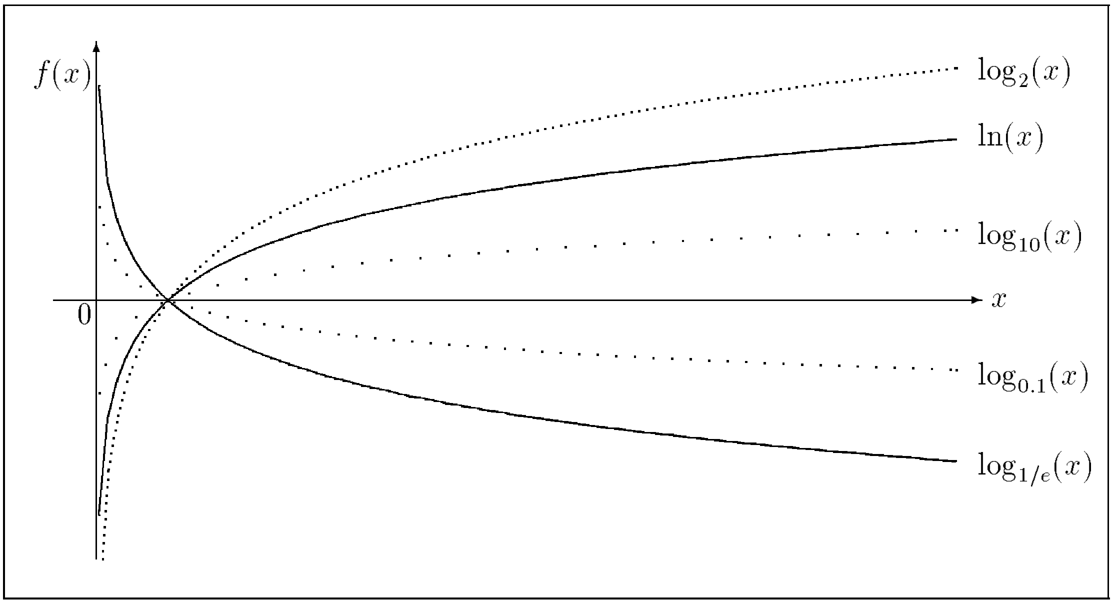


Figure 9.9: Logarithm

It is real only on the positive real axes. In this range it is an increasing function of x. Logarithms to other positive bases b may be written in the form

$$\log_b(x) = \ln(x) \cdot \frac{1}{\ln(b)} \tag{9.43}$$

9.7.1.1 Argument Reduction for $\log_2$

To simplify range reduction most logarithm subroutines on binary machines actually compute $\log_2(x)$, and then form the natural logarithm by an additional multiplication

$$\ln(x) = \log_2(x) \cdot \ln(2) \tag{9.44}$$

Writing

$$x = m \cdot 2^n, \tag{9.45}$$

where n is the integer floating-point exponent of x and m its mantissa with

$$\frac{1}{2} \leq m < 1, \tag{9.46}$$

yield

$$\log_2(x) = n + \log_2(m). \tag{9.47}$$

To approximate $\log_2(m)$ we may use

$$\log_2(m) = t \sum_{k=0}^{\infty} q_k \cdot t^{2k}, \ m > 0 \tag{9.48}$$

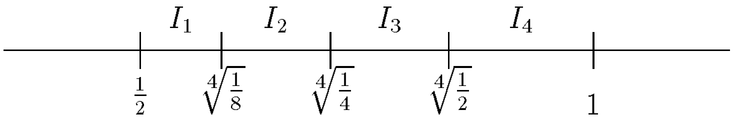


Figure 9.10: Argument Reduction for $\log_2$

with

$$t := \frac{m-1}{m+1}, \quad \text{and} \quad q_k := \frac{2}{(2k+1)\ln 2} \tag{9.49}$$

The range reduction into the basic interval $[1/2, 1]$ indicated by (9.45) and (9.46) requires an approximating function

$$p_N(t) := t \cdot \sum_{k=0}^{N} q_k \cdot t^{2k} \tag{9.50}$$

with N being relative large. Therefore, a further refinement of the reduction process is done. We proceed as follows: The quantity m ranges from $1/2$ to 1. This range is now subdivided into 4 parts $I_1, \ldots, I_4$ according to

$$\begin{array}{lrcl} & I_1 & := & [\frac{1}{2}, \sqrt[4]{\frac{1}{8}}] \quad , \quad I_2 \;:=\; [\sqrt[4]{\frac{1}{8}}, \sqrt[4]{\frac{1}{4}}], \\ & I_3 & := & [\sqrt[4]{\frac{1}{4}}, \sqrt[4]{\frac{1}{2}}] \quad , \quad I_4 \;:=\; [\sqrt[4]{\frac{1}{2}}, 1] \\ \text{i.e.} & I_j & := & [\sqrt[4]{\frac{1}{2^{4-(j-1)}}}, \sqrt[4]{\frac{1}{2^{4-j}}}] \quad , \quad j = 1, 2, 3, 4. \end{array} \tag{9.51}$$

The intervals $I_j,\ j = 1, 2, 3$ are mapped into I_4 by multiplying $m \in I_j$ with

$$c_j := \sqrt[4]{2^{4-j}} \tag{9.52}$$

Thus

$$\begin{aligned} \log_2(m) &= \log_2(m \cdot c_j \cdot \frac{1}{c_j}) \\ &= \log_2(m \cdot c_j) + \log_2 \frac{1}{c_j} \\ &= \log_2(m \cdot c_j) - \log_2 \sqrt[4]{2^{4-j}} \\ &= \log_2(m \cdot c_j) - \frac{4-j}{4} \end{aligned} \tag{9.53}$$

The product $r := m \cdot c_j \in [\sqrt[4]{1/2}, 1]$ is transformed into the reduced argument

$$t = t(r) := \frac{r-1}{r+1} = \frac{m \cdot 2^{\frac{4-j}{4}} - 1}{m \cdot 2^{\frac{4-j}{4}} + 1} = \frac{m - 2^{\frac{j-4}{4}}}{m + 2^{\frac{j-4}{4}}} \tag{9.54}$$

The function $t(r)$ is monotone increasing in r ($r \in [\sqrt[4]{1/2}, 1]$). This leads to the basic interval

$$t([\sqrt[4]{1/2}, 1]) = [\frac{\sqrt[4]{1/2} - 1}{\sqrt[4]{1/2} + 1}, 0] \subseteq [-0.0865, 0].$$

9.7.1.2 Approximation Within the Basic Interval

Within this interval the relative approximation error is bounded by

$$\begin{aligned}
\left| \frac{t \sum_{k=0}^{\infty} q_k \cdot t^{2k} - t \sum_{k=0}^{N} q_k \cdot t^{2k}}{t \sum_{k=0}^{\infty} q_k \cdot t^{2k}} \right| &= \left| \frac{\sum_{k=N+1}^{\infty} q_k \cdot t^{2k}}{\sum_{k=0}^{\infty} q_k \cdot t^{2k}} \right| \\
&\le \frac{1}{q_0} \cdot q_{N+1} \cdot t^{2N+2} \left\{ 1 + \underbrace{\frac{q_{N+2}}{q_{N+1}}}_{\in(0,1)} t^2 + \underbrace{\frac{q_{N+3}}{q_{N+1}}}_{\in(0,1)} t^4 + \ldots \right\} \\
&\le \frac{1}{q_0} \cdot q_{N+1} \cdot t^{2N+2} \cdot \frac{1}{1-t^2} \\
&\le \frac{\ln 2}{2} \cdot \frac{2}{(2N+3)\ln 2} \cdot t^{2N+2} \cdot \frac{1}{1-t^2} = \frac{1}{2N+3} \cdot \frac{t^{2N+2}}{1-t^2} \\
&\overset{|t|<0.0865}{\le} \frac{1.1}{2N+3} (0.0865)^{2N+2} \overset{N=7}{<} 0.636 \cdot 10^{-18} =: \varepsilon(app)
\end{aligned} \tag{9.55}$$

Putting all together shows

$$\log_2(x) \quad = \quad n - \frac{4-j}{4} + t \cdot p(t^2) \cdot (1 + \varepsilon_{\text{app}}) \tag{9.56}$$

with

$$t := \frac{m \cdot c_j - 1}{m \cdot c_j + 1} = \frac{m - c_j}{m + c_j} \tag{9.57}$$

if $m \in I_j$, and $x = m \cdot 2^n$, and an appropriate quantity $\varepsilon_{\text{app}} \in [-\varepsilon(\text{app}), \varepsilon(\text{app})]$. For arguments $x \in [1, \sqrt[4]{2}]$ the argument reduction process would give
$m \in [1/2, \sqrt{1/8}],\ n = 1,\ j = 1$ with $\log_2 x = 1 - 3/4 + t \cdot p(t^2) = 0.75 - t \cdot p(t^2)$.

Thus, for x very near to 1 ($\Rightarrow \log_2 x$ close to 0) cancellation of leading bits in $t \cdot p(t^2)$ will occur. In general, the machine approximation of $t \cdot p(t^2)$ will be contaminated by rounding errors. These errors will be amplified dramatically. Therefore, the case $x \in [1, \sqrt[4]{2})$ is handeled separately. In these case we do not separate x into its mantissa m and its exponent n. Instead

$$\log_2(x) = \frac{x-1}{x+1} \cdot p_\infty((\frac{x-1}{x+1})^2) \approx \frac{x-1}{x+1} \cdot p_N((\frac{x-1}{x+1})^2) \tag{9.58}$$

is used directly.

The value $\dfrac{x-1}{x+1}$ ranges from 0 to $\dfrac{\sqrt[4]{2}-1}{\sqrt[4]{2}+1} < 0.0865$. Therefore, error estimation (9.55) is valid again.

9.7.1.3 Result Adaptation

If $x \in [1/\sqrt[4]{2}, \sqrt[4]{2}]$ formula (9.58) is used and no reconstruction is necessary. In all other cases the representation (9.58) is used, i.e. $\log_2(x)$ is computed via

$$\log_2(x) = n - \frac{4-j}{4} + t \cdot p_\infty(t^2) \approx n - \frac{4-j}{4} + t \cdot p_N(t^2)$$

n according to (9.45), (9.46), j according to (9.53) and t according to (9.54).

9.7.1.4 Algorithms

The argument reduction is done in the following way:

Algorithm 9.16: $\mathsf{ArgRed}\,(x, [r], n, j)$ {procedure}

if $\sqrt[4]{\frac{1}{2}} < x < \sqrt[4]{2}$ **then** $\{x$ near to $1\}$

  $[r] := \dfrac{x-1}{x+1}$

  $n := 0$

  $j := 0$

else

  $m := \mathrm{mant}(x)$ $\{\in [0.5, 1)\}$

  $n := \mathrm{expo}(x)$ $\{x = m \cdot 2^n\}$

  if $m > \sqrt[4]{\frac{1}{4}}$ **then**

    if $m < \sqrt[4]{\frac{1}{2}}$ **then** $j = 3$

    else $j = 4$

  else

    if $m < \sqrt[4]{\frac{1}{8}}$ **then** $j := 1$

    else $j := 2$

$[h] := m \cdot [c_j]$ $\{2^{\frac{4-j}{4}} \in [c_j]\}$

$[r] := \dfrac{[h]-1}{[h]+1}$

return $[r], n, j$

Enclosures for $\log_2(x)$ for point arguments x can be computed via

Algorithm 9.17: LnSimp (x) {function}

if $x \leq 0$ **then** $\{q_k = \dfrac{2}{(2k+1)\ln 2} \in [q_k]\}$
  ERROR
else if $x = 1$ **then** $[s] := 0$
else
  ArgRed $(x, [r], n, j)$
  $[s] := [q_N]$
  for $k := N-1$ **downto** 0 **do**
    $[s] := [s][r]^2 + [q_k]$
  $[s] := [r][s][1-\varepsilon(\text{app}), 1+\varepsilon(\text{app})]$
  $[s] := [s] + n - \dfrac{4-j}{4}$

return LnSimp $:= [s]$

Enclosures for $\log_2([\underline{x}, \overline{x}])$ for interval arguments can be computed by

Algorithm 9.18: LnSimp $([x])$ {function}

if $\text{diam}[x] = 0$ **then** {point argument}
  $[h] :=$ LnSimp $(\underline{x})$ {enclosure of $\ln(\underline{x}) = \ln(\overline{x}) = \ln[x]$}
else
  $[a] :=$ LnSimp $(\underline{x})$ {enclosure of $\ln(\underline{x})$}
  $[b] :=$ LnSimp $(\overline{x})$ {enclosure of $\ln(\overline{x})$}
  $[h] := [\underline{a}, \overline{b}]$ {enclosure of $[\ln(\underline{x}), \ln(\overline{x})]$}

return LnSimp $:= [h]$

9.7.1.5 Program Code

We now give the PASCAL-XSC source code for the module `lnm`. Here the simple program version `LnSimp` which computes an interval enclosure of $\ln(x)$ is implemented.

```
{----------------------------------------------------------------------}
MODULE lnm;  { Logarithm Module Simple Version    }

USE i_ari;   { Make available interval arithmetic }
```

```
CONST
  LnEps = 0.7e-18;  { Bound for the approximation error of ln(x) }
  MaxDegree = 7;    { Degree of the polynomial approximation     }

VAR
  q: ARRAY[0..MaxDegree] OF interval; { Polynomial coefficients for ln  }
  c: ARRAY[1..4] OF real;             { Reduction constants (points)    }
  LncInv: ARRAY[1..4] OF interval;    { Constants for result adaptation }
  OneDownUp: interval;                { [1-LnEps, 1+LnEps]  }
  NearOne:   interval;                { [1, sqrt4(2) ]      }
  Ln2: interval;                      { Enclosure for ln(2) }

{----------------------------------------------------------------------}
{  Argument reduction                                                  }
{----------------------------------------------------------------------}
PROCEDURE ArgRed(x: real; VAR r: interval; VAR Exponx, j: integer);
VAR h: interval;
    Mantx: real;
BEGIN
  IF x IN NearOne THEN BEGIN
    r:= (intval(x) - 1)/(intval(x) + 1);
    { Choose values Exponx and j to skip the reconstruction process: }
    Exponx:= 0;
    j:= 4;
  END
  ELSE BEGIN
    Mantx := mant(x);   { Mantissa in the range [0.5, 1)  }
    Exponx:= expo(x);   { Corresponding exponent          }
    IF Mantx > c[2] THEN BEGIN
      IF Mantx < c[3] THEN
        j:= 3
      ELSE
        j:= 4;
    END
    ELSE BEGIN
      IF Mantx < c[1] THEN
        j:= 1
      ELSE
        j:= 2;
    END;
    r:= ( Mantx - c[j] ) / ( Mantx + c[j] ); { reduced argument }
  END;
END;

{----------------------------------------------------------------------}
{  Logarithm for point arguments x                                     }
{----------------------------------------------------------------------}
GLOBAL FUNCTION LnSimp(x: real): interval;
VAR
  exponx, j : integer;
  k: integer;
  r, rr, s: interval;

BEGIN
  IF x <= 0 THEN BEGIN
    writeln('ERROR, argument in routine LnSimp out of range!!!');
    r:= x/0.0;   { Force program termination }
  END
  ELSE BEGIN
    IF x = 1 THEN  { Special case ln(x) = 1 }
      s:= intval(0)
    ELSE BEGIN
      ArgRed(x, r, exponx, j);
```

```
      rr:= r*r;
      s:= q[MaxDegree];
      FOR k:= MaxDegree-1 DOWNTO 0 DO BEGIN
        s:= s*rr + q[k]
      END;
      s:= r*s*OneDownUp;  { Take approximation error into account }
      s:= s + Exponx*Ln2 - LncInv[j];         { Result adaptation }
    END;
  END;
  LnSimp:= s
END;

{----------------------------------------------------------------------}
{  Logarithm for interval arguments x                                  }
{----------------------------------------------------------------------}
GLOBAL FUNCTION LnSimp(x: interval): interval;
VAR
  u, v, z: interval;
BEGIN
  IF diam(x)=0 THEN
    z:= LnSimp(x.inf)
  ELSE BEGIN { The logarithm is monotone increasing in x }
    u:= LnSimp(x.inf);
    v:= LnSimp(x.sup);
    z:= intval(u.inf, v.sup);
  END;
  LnSimp:= z;
END;

VAR k: integer; { Local loop variable }
BEGIN
{----------------------------------------------------------------------}
{                     Initialization of constants                      }
{----------------------------------------------------------------------}
  ln2:= ln(intval(2));
  NearOne  := intval( 1, sqrt(sqrt(2)) );
  OneDownUp:= intval( 1 -< LnEps, 1 +> LnEps );

  { Polynomial coefficients: }
  FOR k:= 0 TO MaxDegree DO BEGIN
    q[k]:= intval(2)/(2*k+1);   { Interval enclosures }
  END;
  { Reduction constants: }
  c[1]:= sqrt(sqrt(0.125));     { Points              }
  c[2]:= sqrt(sqrt(0.25));
  c[3]:= sqrt(sqrt(0.5));
  c[4]:= 1;
  { Constants for result adaptation }
  LncInv[1]:= ln(intval(1)/c[1]);       { Interval enclosures }
  LncInv[2]:= ln(intval(1)/c[2]);
  LncInv[3]:= ln(intval(1)/c[3]);
  LncInv[4]:= 0; { ln(1/1) }
END.
{----------------------------------------------------------------------}
```

9.7.1.6 Numerical Examples

The driver program `lnmtst` computes un upper bound for the relative approximation error over a set of randomly chosen point interval arguments. The function `MaxRelErr` is taken from module `fnc_util` (see Appendix D). The random number generator is described in Chapter 8.

```
PROGRAM lnmtst;          { Driver for module lnm }

USE lnm;       { Use Logarithm Module Simple Version          }
USE i_ari;     { Interval Module                              }
USE random;    { Random numbers                               }
USE fnc_util; { Make available functions Max, MaxRelErr, ... }

VAR x: real;
    a, b: real;
    RelErr, RelErrIntrinsic: real;
    ilnx,           { Enclosure using LnSimp                      }
    lnx: interval;  { Enclosure using the intrinsic routine       }
    k, kmax: integer;
BEGIN
  RelErr:= 0;
  RelErrIntrinsic:= 0;
  a:= sqrt(1/maxreal);
  b:= sqrt(maxreal);
  { a and b are chosen in this way to avoid intermediate overflow     }
  {    within the random number generator RandLn (see module random). }
  kmax:= 4000;
  FOR k:= 0 TO kmax DO BEGIN
    IF k MOD 1000 = 0 THEN BEGIN
      writeln('Number of test cases so far: ', k);
      writeln('   Maximum error for LnSimp:       ', RelErr);
      writeln('   Maximum error for intrinsic ln: ', RelErrIntrinsic);
    END;
    x:= RandLn(a, b); { Logarithmically distributed random numbers in }
                      {    the range a..b.                              }
    x:= x*x; { Cover the full range 1/maxreal..maxreal }
    ilnx:= LnSimp(intval(x));   { Simple routine from module lnm }
    lnx := ln(intval(x));       { PASCAL-XSC intrinsic routine   }
    RelErr:= max( RelErr, MaxRelErr(ilnx, ilnx) );
    RelErrIntrinsic:= max( RelErrIntrinsic, MaxRelErr(lnx, lnx) );
    IF ilnx >< lnx THEN BEGIN   { Check empty intersection        }
      writeln('ERROR: Intersection of intrisic function result and ');
      writeln('  the simple version LnSimp is empty!');
      x:= x/0;  { Force program termination }
    END;
  END; { Loop over k }
END.
```

The driver program `lnmtst` produces the following output:

```
Number of test cases so far: 0
   Maximum error for LnSimp:       0.000000000000000E+000
   Maximum error for intrinsic ln: 0.000000000000000E+000
Number of test cases so far: 1000
   Maximum error for LnSimp:       9.122245242598870E-016
   Maximum error for intrinsic ln: 2.217792472835038E-016
Number of test cases so far: 2000
   Maximum error for LnSimp:       9.122245242598870E-016
   Maximum error for intrinsic ln: 2.217792472835038E-016
Number of test cases so far: 3000
   Maximum error for LnSimp:       1.579564658432300E-015
   Maximum error for intrinsic ln: 2.219349668262382E-016
Number of test cases so far: 4000
   Maximum error for LnSimp:       1.579564658432300E-015
   Maximum error for intrinsic ln: 2.219349668262382E-016
```

We see that the error bound for `LnSimp` is about four times larger than the corresponding bound for the intrinsic PASCAL-XSC routine `ln` .

$$\mathrm{x}.\underbrace{\mathrm{xxxxxx}}_{\hat{=}k}\mathrm{x}\cdots\mathrm{x} \qquad x \in \left[\frac{17}{16}, 2\right)$$

Figure 9.11: k taken from x in Binary Representation

9.7.2 Computation of $\ln(x)$ using Table-Lookup

We describe a typical table-lookup algorithm for $\ln(x)$ over the range $[16/17, 2]$.

9.7.2.1 Argument Reduction

If $16/17 \leq x \leq 17/16 = 1.0625$ $\ln(x)$ is computed by a polynomial approximation. This polynomial is designed to approximate $\ln(x)$ on $[16/17, 17/16]$. Otherwise, find the breakpoint

$$c_k := 1 + \frac{k}{64}, \quad k \in \{0,\, 1,\, \ldots,\, 64\} \tag{9.59}$$

such that

$$|x - c_k| \leq \frac{1}{128} \tag{9.60}$$

Calculate the reduced argument r by

$$r := \frac{x - c_k}{x + c_k} \quad \in [-\frac{1}{256}, \frac{1}{256}] \tag{9.61}$$

The result of the argument reduction is the index $k \in \{0, \ldots, 64\}$ according to (9.59) in combination with (9.60) as well as the quantity r lying in the basic range $[-2^{-8}, 2^{-8}]$.
Notice that

$$\frac{1+r}{1-r} = \frac{x}{c_k} \quad \in [\frac{255}{257}, \frac{257}{255}] \tag{9.62}$$

holds. In (9.60) the index k may be computed by extracting the first 6 bits after the binary point of x.
We compute the appropriate index k by

$$\left.\begin{array}{rcl} h & := & \mathrm{frac}(x) \\ k & := & \mathrm{trunc}(64 \cdot h) \\ z & := & x - (1 + \dfrac{k}{64}) \qquad (\hat{=} x - c_k) \\ \textbf{if} \ z > \dfrac{1}{118} & \textbf{then} & k := k + 1 \end{array}\right\} \tag{9.63}$$

Now it holds

$$|x - (1 + \frac{k}{64})| \leq \frac{1}{128}.$$

9.7.2.2 Approximation Within the Basic Range

In the basic range $r \in [-2^{-8}, 2^{-8}]$ the representation

$$\begin{aligned} \ln(\frac{x}{c_k}) &\overset{(9.62)}{=} \ln\frac{1+r}{1-r} \\ &= r \cdot \sum_{k=0}^{\infty} \frac{2}{2k+1} \cdot r^{2k} =: r \cdot p_\infty(r^2) \end{aligned} \tag{9.64}$$

is exploited. Truncating this series after $n+1$ summands is accompanied by the approximation error

$$\left| \frac{\log_2 \frac{1+r}{1-r} \cdot r \cdot p_n(r^2)}{\log_2 \frac{1+r}{1-r}} \right| = \frac{\sum\limits_{k=n+1}^{\infty} \frac{1}{2k+1} \cdot r^{2k}}{\sum\limits_{k=0}^{\infty} \frac{1}{2k+1} \cdot r^{2k}}$$

$$\begin{aligned} \frac{1}{2n+3} \cdot r^{2n+2} \sum_{k=n+1}^{\infty} \frac{2n+3}{2k+1} \cdot r^{2k-(2n+2)} &\leq \frac{1}{2n+3} \cdot \frac{1}{1-r^2} \cdot r^{2n+2} \\ &\leq 5.08 \cdot 10,\ n := 2, |r| \leq 2^{-8} \end{aligned}$$

I.e. the polynomial

$$r \cdot p_2(r^2) = r \cdot (2 + \frac{2}{3} \cdot r^2 + \frac{2}{5} \cdot r^4) \tag{9.65}$$

approximates $\ln(x/c_k)$ in the basic interval with a relative error bound less than $5.08 \cdot 10^{-16}$. An alternative method would be to compute a best approximating polynomial on the basic range using the algorithm of Remez (see ???).
For $x \in [16/17, 17/16]$ an argument reduction is applied which does not imply an additive reconstruction process. We simply use

$$\ln(x) = \ln\frac{1+r}{1-r} = r \cdot p_\infty(r^2)\,.$$

With $r := (x-1)/(x+1) \in [-1/33, 1/33]$ and $n := 4$ we find as bound for the relative approximation error

$$\left| \frac{\ln(x) - r \cdot p_4(r^2)}{\ln(x)} \right| < \frac{1.01}{???} \cdot (\frac{1}{33})^{10} < 6 \cdot 10^{???} \tag{9.66}$$

9.7.2.3 Result Adaptation

The reconstruction process uses

$$\ln(x) = \ln(c_k) + \ln(\frac{x}{c_k}) \overset{(9.64)}{=} \ln(c_k) + r \cdot p_\infty(r^2) \tag{9.67}$$

The quantities

$$d_k := \ln(c_k),\ k = 0, 1, \ldots, 64 \tag{9.68}$$

are calculated and stored in a table. To get an interval enclosure, we have to use enclosures $[d_k]$ of the reconstruction constants d_k. It is enough to store $\bigtriangledown d_k$, $k = 0,\ldots,64$. Enclosures $[d_k]$ may be computed by $[d_k] := [\bigtriangledown d_k, \text{succ}\bigtriangledown d_k]$. Of course, this results in a small runtime penalty. It is not necessary to store the quantities $c_k = 1 + k/64$ (see (9.63)).

9.7.2.4 Algorithms

We first give the algorithm ArgRed for the argument reduction process.

Algorithm 9.19: ArgRed $(x, k, [r])$ {subroutine}

$\{c_k := 1 + \frac{k}{64},\ k = 0, 1, \ldots, 64\}$

1. Find $k \in \{0,\ldots,64\}$ such that
   $|x - c_k| \le \frac{1}{128}$

   $\{k$ can be found using the first 6 bits behind the$\}$
   {binary point of x (see page 305) }

2. $[r] := \dfrac{x - c_k}{[x, x] + c_k}$ {enclosure for the reduced argument}
   {computed by interval arithmetic }

return $k, [r]$

Notice, that the quantity $x - c_k$ in (9.63) is computed exactly by ordinary floating-point arithmetic. This is not true for the denominator. Therefore, the denominator is forced to be calculated by interval arithmetic.

The next algorithm `iLn` computes an enclosure of $\ln(x)$ for a point argument $x \in [16/17, 2]$ using table lookup.

Algorithm 9.20: iLn (x) {function}

$\{a_k := \frac{2}{2k+1},\ k = 0, 1\ldots, n;\quad n = 4,\ p_n(r^2) = \sum_{k=0}^{n} a_k r^{2k}\}$

if $x \in [\frac{16}{17}, \frac{17}{16}]$ **then**

$[r] := \dfrac{x - 1}{[x, x] + 1}$ {expression is computed by interval arithmetic }

$[s] := [r] \cdot p_4([r]^2)$ {interval Horner scheme }

$[s] := [x] \cdot [1 - \text{EpsLn1}, 1 + \text{EpsLn1}]$ {approximation error}

else

$\{[d_k] \ni \log_2(c_k) = \log_2(1 + \frac{k}{64}),\ k = 0, 1, \ldots, 64\}$

ArgRed $(x, k, [r])$

$[s] := [r] \cdot p_2([r]^2)$ {interval Horner scheme}
$[s] := [s] \cdot [1 - \text{EpsLn2}, 1 + \text{EpsLn2}]$ {approximation error}
$[s] := [s] + [d_k]$ {result adaptation}

return $\mathsf{iLn} := [s]$

Based on the enclosure algorithm for point arguments the algorithm `iLn` for interval arguments $[x] \subseteq [16/17, 2]$ is now given:

Algorithm 9.21: iLn $([x])$ {function}

1. **if** $[x] \not\subseteq [\frac{16}{17}, 2]$ **then**

    ERROR
    STOP
  else
    if $\text{diam}([x]) = 0$ **then**
      $[z] := \mathsf{iLn}\,(\underline{x})$
    else
      $[u] := \mathsf{iLn}\,(\underline{x})$
      $[v] := \mathsf{iLn}\,(\overline{x})$
      $[z] := [\underline{u}, \overline{v}]$

  return $\mathsf{iLn} := [z]$

9.7.2.5 Program Code

We now give the PASCAL-XSC source code for the module `lnmi`. Here the table-driven program version `iLn` which computes an interval enclosure of $\ln([x])$ is implemented.

```
MODULE lnmi;  { Logarithm Module Improved Version   }

USE i_ari;    { Make available interval arithmetic }

CONST
  LnEps1 = 6.0e-17;  { Bound for the approximation error of ln(x), }
                     {   x in [16/17, 17/16] }
  LnEps2 = 5.08e-16; { Bound for the approximation error of ln(x), }
                     {   x in (17/16, 2]     }


VAR                           { Global variables }
  a: ARRAY[0..4] OF interval; { Polynomial coefficients for ln }
  d: ARRAY[0..64] OF real;    { Constants for reconstruction   }
  t1,                         { 16/17 }
  t2: real;                   { 17/16 }
  Inv64: real;                { 1/64  }
  Inv128: real;               { 1/128 }
```

```
{-----------------------------------------------------------}
{ Compute an interval enclosure of the reduced argument.    }
{   x is an element of (17/16, 2]                           }
{-----------------------------------------------------------}
PROCEDURE ArgRed(x: real; VAR k: integer; VAR r: interval);
VAR
  c, z: real;
BEGIN
  k:= trunc( 64*(x - 1) );
  c:= 1 + k*Inv64;
  z:= x - c;
  IF z > Inv128 THEN BEGIN
    k:= k + 1;
    c:= c + Inv64;
  END;
  r:= (intval(x) - c) / (intval(x) + c);
END;


FUNCTION LnLook(x: real): interval;
VAR
  k: integer;
  r, rr, s: interval;
BEGIN
  IF x <= t2 THEN BEGIN  { x in [16/17, 17/16] }
    r:= (intval(x) - 1.0) / (intval(x) + 1.0);
    rr:= r*r;
    s:= r*(((( a[4]*rr + a[3] )*rr + a[2] )*rr + a[1] )*rr + a[0] );
    s:= s*intval(1.0 -< LnEps1, 1.0 +> LnEps1);
  END
  ELSE BEGIN   { x is an element of (17/16, 2] }
    ArgRed(x, k, r);                    { Reduction      }
    rr:= r*r;
    s:= r*( ( a[2]*rr + a[1] )*rr + a[0] );
    s:= s*intval(1.0 -< LnEps2, 1.0 +> LnEps2);
    s:= intval(d[k], succ(d[k])) + s;   { Reconstruction }
  END;
  LnLook:= s;
END;


GLOBAL FUNCTION iln(x: interval): interval;
VAR
  u, v, z: interval;
BEGIN
  IF NOT (x <= intval(t1, 2.0)) THEN BEGIN   { x not in [16/17,2] }
    writeln('ERROR: Argument ', x, ' not in [ 16/17 , 2 ]');
    z:= x;
  END
  ELSE IF diam(x) = 0 THEN { Point argument }
    z:= LnLook(x.inf)
  ELSE BEGIN               { Proper interval argument }
    u:= LnLook(x.inf);
    v:= LnLook(x.sup);
    z:= intval(u.inf, v.sup);
  END;
  iln:= z;
END;


BEGIN    { Initialization part }
  Inv64 := 1.0/64;
  Inv128:= 1.0/128;
  t1:=   8477364004462111.0 / power(2,53);  { 9.4117647E-001 = 16/17 }
```

```
t2:=   4785074604081152.0 / power(2,52);  { 1.0625000E+000 = 17/16 }

a[0].inf:=   4503599627370496.0 / power(2,51);  { 2.0000000E+000 }
a[1].inf:=   6004799503160661.0 / power(2,53);  { 6.6666667E-001 }
a[2].inf:=   7205759403792793.0 / power(2,54);  { 4.0000000E-001 }
a[3].inf:=   5146971002709138.0 / power(2,54);  { 2.8571429E-001 }
a[4].inf:=   8006399337547548.0 / power(2,55);  { 2.2222222E-001 }
a[0].sup:=   4503599627370496.0 / power(2,51);  { 2.0000000E+000 }
a[1].sup:=   6004799503160662.0 / power(2,53);  { 6.6666667E-001 }
a[2].sup:=   7205759403792794.0 / power(2,54);  { 4.0000000E-001 }
a[3].sup:=   5146971002709139.0 / power(2,54);  { 2.8571429E-001 }
a[4].sup:=   8006399337547549.0 / power(2,55);  { 2.2222222E-001 }

d[ 0]:=                  0.0 ;                { 0.0000000E+000 }
d[ 1]:=   8937555034375139.0 / power(2,59);  { 1.5504187E-002 }
d[ 2]:=   8869326752330496.0 / power(2,58);  { 3.0771659E-002 }
d[ 3]:=   6601849900817660.0 / power(2,57);  { 4.5809536E-002 }
d[ 4]:=   8736928775103142.0 / power(2,57);  { 6.0624622E-002 }
d[ 5]:=   5420418749682128.0 / power(2,56);  { 7.5223421E-002 }
d[ 6]:=   6457236551723851.0 / power(2,56);  { 8.9612159E-002 }
d[ 7]:=   7479347221550435.0 / power(2,56);  { 1.0379679E-001 }
d[ 8]:=   8487162167882469.0 / power(2,56);  { 1.1778304E-001 }
d[ 9]:=   4740537887264345.0 / power(2,55);  { 1.3157636E-001 }
d[10]:=   5230733163492643.0 / power(2,55);  { 1.4518201E-001 }
d[11]:=   5714348438420766.0 / power(2,55);  { 1.5860503E-001 }
d[12]:=   6191558024467411.0 / power(2,55);  { 1.7185026E-001 }
d[13]:=   6662529397872905.0 / power(2,55);  { 1.8492234E-001 }
d[14]:=   7127423551559348.0 / power(2,55);  { 1.9782574E-001 }
d[15]:=   7586395325513714.0 / power(2,55);  { 2.1056477E-001 }
d[16]:=   8039593716390433.0 / power(2,55);  { 2.2314355E-001 }
d[17]:=   8487162167882469.0 / power(2,55);  { 2.3556607E-001 }
d[18]:=   8929238843276842.0 / power(2,55);  { 2.4783616E-001 }
d[19]:=   4682978440745187.0 / power(2,54);  { 2.5995752E-001 }
d[20]:=   4898722318886414.0 / power(2,54);  { 2.7193372E-001 }
d[21]:=   5111912955083109.0 / power(2,54);  { 2.8376817E-001 }
d[22]:=   5322610076359984.0 / power(2,54);  { 2.9546421E-001 }
d[23]:=   5530871338190313.0 / power(2,54);  { 3.0702504E-001 }
d[24]:=   5736752419200706.0 / power(2,54);  { 3.1845373E-001 }
d[25]:=   5940307110524973.0 / power(2,54);  { 3.2975329E-001 }
d[26]:=   6141587400165834.0 / power(2,54);  { 3.4092659E-001 }
d[27]:=   6340643552695470.0 / power(2,54);  { 3.5197642E-001 }
d[28]:=   6537524184600582.0 / power(2,54);  { 3.6290549E-001 }
d[29]:=   6732276335554499.0 / power(2,54);  { 3.7371641E-001 }
d[30]:=   6924945535877722.0 / power(2,54);  { 3.8441170E-001 }
d[31]:=   7115575870428922.0 / power(2,54);  { 3.9499381E-001 }
d[32]:=   7304210039150667.0 / power(2,54);  { 4.0546511E-001 }
d[33]:=   7490889414477897.0 / power(2,54);  { 4.1582790E-001 }
d[34]:=   7675654095802210.0 / power(2,54);  { 4.2608440E-001 }
d[35]:=   7858542961171323.0 / power(2,54);  { 4.3623677E-001 }
d[36]:=   8039593716390433.0 / power(2,54);  { 4.4628710E-001 }
d[37]:=   8218842941680597.0 / power(2,54);  { 4.5623743E-001 }
d[38]:=   8396326136038560.0 / power(2,54);  { 4.6608973E-001 }
d[39]:=   8572077759432583.0 / power(2,54);  { 4.7584590E-001 }
d[40]:=   8746131272959724.0 / power(2,54);  { 4.8550782E-001 }
d[41]:=   8918519177081630.0 / power(2,54);  { 4.9507727E-001 }
d[42]:=   4544636524024062.0 / power(2,53);  { 5.0455601E-001 }
d[43]:=   4629211786305344.0 / power(2,53);  { 5.1394575E-001 }
d[44]:=   4713000290560642.0 / power(2,53);  { 5.2324814E-001 }
d[45]:=   4796016539552918.0 / power(2,53);  { 5.3246480E-001 }
d[46]:=   4878274638697961.0 / power(2,53);  { 5.4159728E-001 }
d[47]:=   4959788310448494.0 / power(2,53);  { 5.5064712E-001 }
d[48]:=   5040570908033232.0 / power(2,53);  { 5.5961579E-001 }
d[49]:=   5120635428585282.0 / power(2,53);  { 5.6850474E-001 }
d[50]:=   5199994525692186.0 / power(2,53);  { 5.7731537E-001 }
```

```
    d[51]:=   5278660521397899.0 / power(2,53);  { 5.8604905E-001 }
    d[52]:=   5356645417685182.0 / power(2,53);  { 5.9470711E-001 }
    d[53]:=   5433960907465171.0 / power(2,53);  { 6.0329085E-001 }
    d[54]:=   5510618385099289.0 / power(2,53);  { 6.1180154E-001 }
    d[55]:=   5586628956477178.0 / power(2,53);  { 6.2024041E-001 }
    d[56]:=   5662003448672942.0 / power(2,53);  { 6.2860866E-001 }
    d[57]:=   5736752419200706.0 / power(2,53);  { 6.3690746E-001 }
    d[58]:=   5810886164889275.0 / power(2,53);  { 6.4513796E-001 }
    d[59]:=   5884414730394544.0 / power(2,53);  { 6.5330127E-001 }
    d[60]:=   5957347916367274.0 / power(2,53);  { 6.6139848E-001 }
    d[61]:=   6029695287292825.0 / power(2,53);  { 6.6943065E-001 }
    d[62]:=   6101466179018540.0 / power(2,53);  { 6.7739882E-001 }
    d[63]:=   6172669705983605.0 / power(2,53);  { 6.8530400E-001 }
    d[64]:=   6243314768165359.0 / power(2,53);  { 6.9314718E-001 }
  END.
```

The driver program `lnmitst` computes un upper bound for the relative approximation error over a set of randomly chosen point interval arguments. The function `MaxRelErr` is taken from module `fnc_util` (see Appendix D). The random number generator is described in Chapter 8.

```
PROGRAM lnmitst;          { Driver for module lnmi }

USE lnmi;     { Use Logarithm Module Simple Version }
USE i_ari;    { Interval Module                   }
USE random;   { Random numbers                    }
USE x_real;   { Make available hexadecimal output of flp-numbers }
USE fnc_util; { Make available functions Max, MaxRelErr, ...     }

VAR x: real;
    BadArg: real;
    a, b: real;
    RelErr, RelErrIntrinsic: real;
    iLnx,          { Enclosure using iLn from module lnmi     }
    lnx: interval; { Enclosure using the intrinsic routine    }
    k, kmax: integer;
BEGIN
  RelErr:= 0;
  RelErrIntrinsic:= 0;
  BadArg:= 1;
  a:= 16/17;   { Valid range for the table look-up approximation }
  b:= 2;       {   is [16/17, 2]. See module lnmi.               }
  kmax:= 30000;
  FOR k:= 0 TO kmax DO BEGIN
    IF k MOD 10000 = 0 THEN BEGIN
      writeln('Number of test cases so far: ', k);
      writeln('   Maximum error for iLn:          ', RelErr);
      writeln('   Maximum error for intrinsic ln: ', RelErrIntrinsic);
    END;
    x:= Rand(a, b); { Uniformly distributed random numbers in }
                    {   the range a..b.                       }
    iLnx:= iLn(intval(x));   { Routine from module lnmi     }
    lnx := ln(intval(x));    { PASCAL-XSC intrinsic routine }
    IF MaxRelErr(iLnx, lnx) > RelErr THEN BEGIN
      { Notice: As denominator in the expression for the relative  }
      {   error we use the better enclosure lnx computed via the   }
      {   intrinsic PASCAL-XSC routine.                            }
      BadArg:= x;                         { Update worst argument }
      RelErr:= MaxRelErr(iLnx, lnx)     { Update worst error bound }
    END;
    RelErrIntrinsic:= max( RelErrIntrinsic, MaxRelErr(lnx, lnx) );
    IF ilnx >< lnx THEN BEGIN   { Check empty intersection       }
```

```
        writeln('ERROR: Intersection of intrisic function result and ');
        writeln('  the simple version LnSimp is empty!');
        x:= x/0;  { Force program termination }
      END;
    END; { Loop over k }
    writeln('BadArg: ', BadArg, '  hexadecimal: ', BadArg:'X');
    ilnx:= iLn(intval(BadArg));   { Routine from module lnmi     }
    lnx := ln(intval(BadArg));    { PASCAL-XSC intrinsic routine }
    writeln('   Error for iLn:          ', MaxRelErr(iLnx,lnx) );
    writeln('   Error for intrinsic ln: ', MaxRelErr(lnx,lnx) );
    writeln('Hexadecimal representation of inf(iLn) and sup(iLn): ');
    writeln( '   ', inf(iLnx):'X' );
    writeln( '   ', sup(iLnx):'X' );
    writeln('Hexadecimal representation of inf(ln) and sup(ln): ');
    writeln( '   ', inf(lnx):'X' );
    writeln( '   ', sup(lnx):'X' );
  END.
```

The driver program `lnmitst` produces the following output:

```
Number of test cases so far: 0
   Maximum error for iLn:           0.000000000000000E+000
   Maximum error for intrinsic ln:  0.000000000000000E+000
Number of test cases so far: 10000
   Maximum error for iLn:           1.109472094051167E-015
   Maximum error for intrinsic ln:  4.410181866340259E-016
Number of test cases so far: 20000
   Maximum error for iLn:           1.329172925403734E-015
   Maximum error for intrinsic ln:  4.410181866340259E-016
Number of test cases so far: 30000
   Maximum error for iLn:           1.329172925403734E-015
   Maximum error for intrinsic ln:  4.410181866340259E-016
BadArg:  1.031818479013052E+000  hexadecimal: 3FF0825417EC4FF3
   Error for iLn:           1.329172925403734E-015
   Error for intrinsic ln:  2.215288209006223E-016
Hexadecimal representation of inf(iLn) and sup(iLn):
   3FA00989646B49F5
   3FA00989646B49FC
Hexadecimal representation of inf(ln) and sup(ln):
   3FA00989646B49F6
   3FA00989646B49F7
```

We see that the error bound for `iLn` is about four times larger than the corresponding bound for the intrinsic PASCAL-XSC routine `ln` . To improve the error bound for the table lookup method some constants have to be stored in higher precision (for example as unevaluated sums of two floating-point numbers). Such accurate table-driven routines are described in [314, 315, 316]. To avoid intermediate computations in a higher precision data format, at some rare points a higher precision arithmetic has to be simulated by ordinary IEEEE arithmetic. The determination of reliable a priori error bouds using interval computations may be found in [135].

9.8 The Power Function $[x]^{[y]}$

The power function $\mathrm{pwr}([x],[y]) := [x]^{[y]}$ is treated as a function depending on two real interval variables x and y. To find an enclosure of the set $R := \{a^b | a \in$

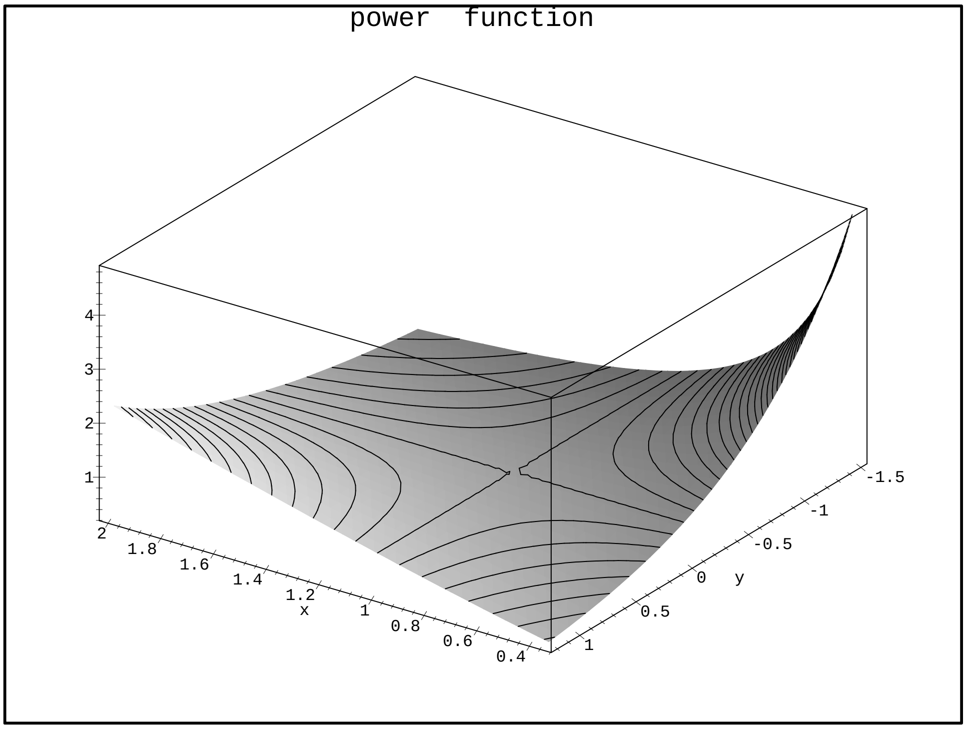


Figure 9.12: Power Function

$[x]$ and $b \in [y]\}$ the extremal point for the minimum function value as well as the extremal point for the maximal function value have to be determined (see Table 9.2). Each extremal value is given as a pair of real values. To find the final inclusion of R the extremal values corresponding to the extremal points are enclosed by intervals. The convex hull of these intervals is a superset of the range of function R for the actual interval arguments x and y.

9.8.1 Power Function for Point Arguments

The general formula to compute x^y is

$$x^y = \exp(y \cdot \ln(x)).$$

This formula only hold for arguments $x > 0$. In the case $x < 0$ (in this case y must be an integer value) the power x^y is computed by

$$x^y = \pm \exp(y \cdot \ln(\text{abs}(x))).$$

The positive sign holds for even values of y, the negative for odd values of y. In the case $x = 0$ the function result $x^y = 1$ is returned.
Using interval versions for the exponential function as well as for the natural logarithm inclusions for the power function for point arguments may be computed as shown in Algorithm 1.1.

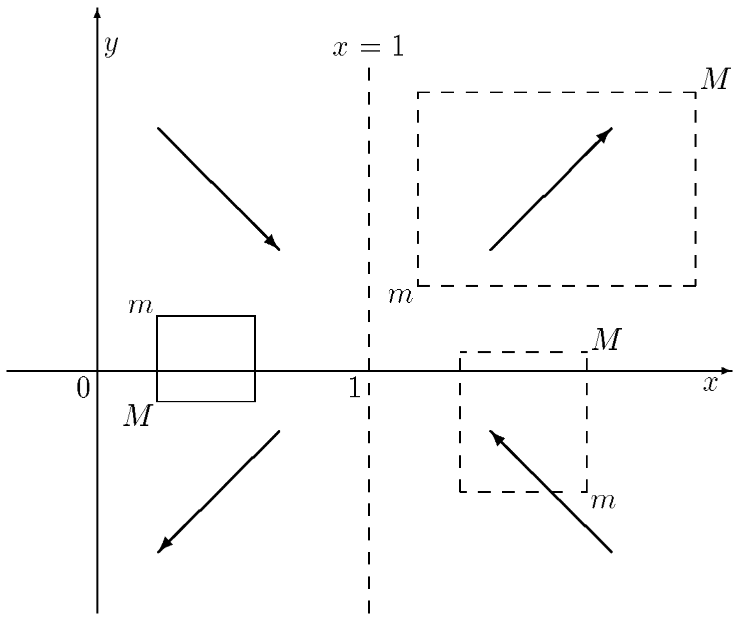


Figure 9.13: Monotonicity behaviour of pwr(x, y)

9.8.2 Power Function for Interval Arguments

To handle interval arguments the monotonicity behaviour of x^y with respect to both variables has to be considered. To find the monotonicity behaviour the partial derivatives with respect to x and y may be used. It holds

$$\frac{\partial x^y}{\partial x} = \frac{\partial e^{y\cdot\ln(x)}}{\partial x} = \frac{y}{x}\, e^{y\cdot\ln(x)}\ ,$$

$$\frac{\partial x^y}{\partial y} = \ln(x)\cdot e^{y\cdot\ln(x)}.$$

The partial derivative with respect to x is positive in the upper right half plane $x > 0,\ y > 0$ and negative in the lower right half plane. The partial derivative with respect to y is negative in the range $0 < x < 1, y \in I\!R$; it is positive for $x \in (1, \infty), y \in I\!R$. Consequently, the extremal curves are the $x-$axis $y = 0$ and the vertical line $x = 1$. On these lines one of the partial derivatives is zero. The monotonicity behaviour of the power function is summarized in Figure 9.13. The extremal points are indicated by m and M, respectively.
Table 9.2 shows a complete list of all different cases including values of $\inf(x) \le 0$. The notation $y \in Z\!\!Z$ means that y is a point interval with $y_1 = y_2 \in Z\!\!Z$. The lower bound of the final result is computed in the form $w_1 := \text{pwr}(u, v)$ with u, v being the entries of the column of Table 9.2 indicated by w_1. The upper bound is computed using the entries of the column indicated by w_2. For example $[0, 0.5] ** [0.1, 0.2]$ leads to case 7 and therefore, to the resulting interval

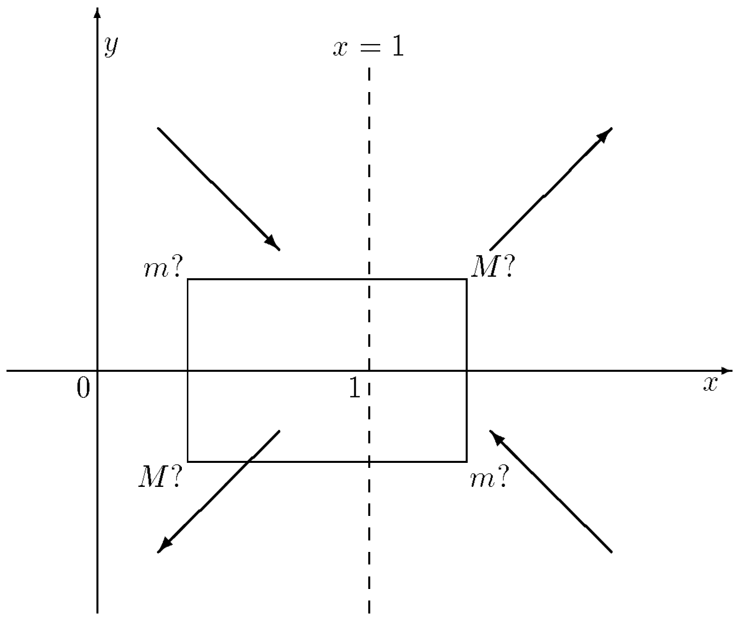


Figure 9.14: pwr($[x]$, $[y]$) with $1 \in [x]$ and $0 \in [y]$

$w = [0, \mathrm{pwr}(x_2, y_2)] = [0.5, \mathrm{pwr}(0.5, 0.2)]$. An error occurs if

- Err1: $0 \in x$; and y is an integer < 0 ,
- Err2: $\inf(x) < 0$ and $y \notin \mathbb{Z} \setminus \{0\}$,
- Err3: $\inf(x) = 0$ and $0 > \inf(y)$ with $y \notin \mathbb{Z}$.

The program listing of the following section refers to Table 9.2.

9.8.3 Algorithms

The power function for point arguments x and y. The exact value x^y is an element of the resulting interval returned by the function.

Algorithm 9.22: Pwr (x, y) {function}

    if $x < 0$ **then**
        $[r] := \exp(y \cdot \ln([x, x]))$
        if $y \in \{2k+1 |\ k \in Z\}$ **then** $[r] := -[r]$
    else
        if $x = 0$ **then**

$$w = [w_1,\ w_2] := [x]^{[y]} = [x_1,\ x_2]^{[y_1,\ y_2]}$$

<table>
<tr><th colspan="5">Conditions</th><th>case</th><th>w_1</th><th>w_2</th></tr>
<tr><td colspan="5">$y_1 = y_2 = 0$</td><td>0</td><td>$w_1 = 1$</td><td>$w_2 = 1$</td></tr>
<tr><td rowspan="7">$x_1 < 0$</td><td rowspan="6">$y \in \mathbb{Z}$</td><td rowspan="3">$y_2 \leq -1$</td><td colspan="2">$x_2 \geq 0$</td><td>Err1</td><td colspan="2">Error</td></tr>
<tr><td rowspan="2">$x_2 < 0$</td><td>y_2 even</td><td>1</td><td>$x_1,\ y_1$</td><td>$x_2,\ y_2$</td></tr>
<tr><td>y_2 odd</td><td>2</td><td>$x_2,\ y_1$</td><td>$x_1,\ y_2$</td></tr>
<tr><td rowspan="3">$y_2 \geq 1$</td><td rowspan="2">y_2 even</td><td>$x_2 < 0$</td><td>3</td><td>$x_2,\ y_1$</td><td>x_1, x_2</td></tr>
<tr><td>$x_2 \geq 0$</td><td>4</td><td>$w_1 = 0$</td><td>$\max\{\lvert x_1\rvert,\ \lvert x_2\rvert\},\ y_2$</td></tr>
<tr><td colspan="2">y_2 odd</td><td>5</td><td>$x_1,\ y_1$</td><td>$x_2,\ y_2$</td></tr>
<tr><td colspan="4">$(y_1 \neq y_2)$ or $(y_1 \notin \mathbb{Z} \setminus \{0\})$</td><td>Err2</td><td colspan="2">Error</td></tr>
<tr><td rowspan="4">$x_1 = 0$</td><td colspan="4">$y_1 < 0$</td><td>Err3</td><td colspan="2">Error</td></tr>
<tr><td rowspan="3">$y_1 \geq 0$</td><td rowspan="2" colspan="2">$x_2 \leq 1$</td><td>$y_1 = 0$</td><td>6</td><td>$w_1 = 0$</td><td>$w_2 = 1$</td></tr>
<tr><td>$y_1 > 0$</td><td>7</td><td>$w_1 = 0$</td><td>$x_2,\ y_1$</td></tr>
<tr><td colspan="3">$x_2 > 1$</td><td>8</td><td>$w_1 = 0$</td><td>$x_2,\ y_2$</td></tr>
<tr><td rowspan="9">$x_1 > 0$</td><td rowspan="3">$x_2 \leq 1$</td><td colspan="3">$y_2 \leq 0$</td><td>9</td><td>$x_2,\ y_2$</td><td>$x_1,\ y_1$</td></tr>
<tr><td colspan="3">$y_1 \geq 0$</td><td>10</td><td>$x_1,\ y_2$</td><td>$x_2,\ y_1$</td></tr>
<tr><td colspan="3">$y_1 < 0 < y_2$</td><td>11</td><td>$x_1,\ y_2$</td><td>$x_1,\ y_1$</td></tr>
<tr><td rowspan="3">$x_1 \geq 1$</td><td colspan="3">$y_2 \leq 0$</td><td>12</td><td>$x_2,\ y_1$</td><td>$x_1,\ y_2$</td></tr>
<tr><td colspan="3">$y_1 \geq 0$</td><td>13</td><td>$x_1,\ y_1$</td><td>$x_2,\ y_2$</td></tr>
<tr><td colspan="3">$y_1 < 0 < y_2$</td><td>14</td><td>$x_2,\ y_1$</td><td>$x_2,\ y_2$</td></tr>
<tr><td rowspan="3">$1 \in x$</td><td colspan="3">$y_2 \leq 0$</td><td>15</td><td>$x_2,\ y_1$</td><td>$x_1,\ y_1$</td></tr>
<tr><td colspan="3">$y_1 \geq 0$</td><td>16</td><td>$x_1,\ y_2$</td><td>$x_2,\ y_2$</td></tr>
<tr><td colspan="3">$y_1 < 0 < y_2$</td><td>17</td><td colspan="2">$w_1 = \min\{x_1^{y_2},\ x_2^{y_1}\}$
$w_2 = \max\{x_2^{y_2},\ x_1^{y_1}\}$</td></tr>
</table>

Table 9.2: Interval-Logic for the Power Function

$[r] := 0$
else
$[r] := \exp(y \cdot \ln([x, x]))$

return Pwr:= $[r]$

Here exp and ln refer to interval functions. To emphasize this fact, we use the explicit notation $[x, x]$ to indicate that the real value (point) x is treated as a point interval.
The next algorithm summarizes the computation of the power function for interval arguments.

Algorithm 9.23: Pwr $([x], [y])$ {function}

1. $x_1 := \underline{x},\ x_2 := \overline{x},\ y_1 := \underline{y},\ y_2 := \overline{y}$
2. Compute an enclosure $[w_1] :=$ Pwr(u, v) with $u \in \{x_1,\ x_2\}$ and $v \in \{y_1,\ y_2\}$ being the first and second entry in column 'w_1' of Table 9.2 corresponding to the actual values of $[x]$ and $[y]$.

   If $w_1 = 0$ or $w_1 = 1$ is indicated, make the corresponding assignment (no additional computation has to be performed). In case 17 compute $[w_1]$ by

   $$[w_1] := \text{interval hull}(\mathsf{Pwr}\,(x_1, y_2),\ \mathsf{Pwr}\,(x_2, y_1)).$$

3. Compute an enclosure $[w_2] :=$ Pwr(u, v) with $u \in \{x_1,\ x_2,\ |x_1|,\ |x_2|\}$ and $v \in \{y_1,\ y_2\}$ being the first and second entry in column 'w_2' of Table 9.2. If $w_2 = 1$ is indicated only an assignment is necessary and in case 17 $[w_2]$ is computed by

   $$[w_2] := \text{interval hull}(\mathsf{Pwr}\,(x_1, y_1),\ \mathsf{Pwr}\,(x_2, y_2))$$

4. **if** NO ERROR (see again Table 9.2) **then**

   $[r] := \text{interval hull}([w_1], [w_2])$
5. **return** Pwr:= $[r]$

The interval hull of two intervals $[u]$ and $[v]$ is defined to be

$$\text{intervall hull}\ ([u], [v]) := [\min\{\underline{u}, \underline{v}\},\ \max\{\overline{u}, \overline{v}\}].$$

9.8.4 PASCAL-XSC Program Listings

The following program listing shows the PASCAL-XSC module pwr for the computation of interval enclosures of pwr(x,y).

```
{----------------------------------------------------------------}
{           Interval Power Function: Module pwrm                 }
{----------------------------------------------------------------}
MODULE pwrm;

USE i_ari;    { Interval arithmetic                             }
USE fnc_util; { Make available IsEven, IsInteger, MaxAbs, ... }
VAR
  GLOBAL VAR test: boolean;

FUNCTION error(code: integer): interval;
BEGIN
  writeln('Error code: ', code);
  writeln('****** ERROR:  Invalid arguments for pwr !!!!!');
  error:=  code;  { Dangerous!                                   }
                  {  It would be better to force program termination }
END;
{----------------------------------------------------------------}

FUNCTION p(x, y: real): interval;  { x**y for point arguments }
VAR res: interval;
BEGIN
  IF x < 0.0 THEN BEGIN
    res:= exp(y*ln(intval(abs(x))));
    IF odd(trunc(y)) THEN p:= -res ELSE p:= res;
  END
  ELSE
    IF x = 0.0 THEN
      p:= intval(0.0)
    ELSE
      p:= exp(y*ln(intval(x)));
END;
{----------------------------------------------------------------}

{----------------------------------------------------------------}
{ Power function x**y for real interval arguments (interval logic) }
{----------------------------------------------------------------}
GLOBAL FUNCTION Pwr(x, y: interval): interval;
LABEL 999; { all is done }
VAR
  res: interval;
  x1, x2, y1, y2: real;
  u1, u2, v1, v2, res1, res2: real;

BEGIN
  x1:= x.inf; x2:= x.sup; y1:= y.inf; y2:= y.sup;
  IF (y1 = y2) AND (y1 = 0.0) THEN BEGIN
    Pwr:= intval(1.0);    { [...]**0 = 1              }
    IF test THEN writeln('Case 0');
    GOTO 999; { return }
  END;
  IF x1 < 0.0 THEN BEGIN
    IF (y1 = y2) AND IsInteger(y1) THEN BEGIN
      IF y2 < 0.0 THEN  { y negative integer }
        IF x2 >= 0 THEN BEGIN
          { Case: Err1; y is a negative integer and 0 in x }
          Pwr:= error(1)
        END
        ELSE { x2 < 0 }
          IF IsEven(y2) THEN BEGIN
            IF test THEN writeln('Case 1');
            Pwr:= p(x1,y1) +* p(x2,y2);
          END
          ELSE BEGIN { y2 is an odd number }
```

```
          IF test THEN writeln('Case 2');
          Pwr:= p(x2,y1) +* p(x1,y2);
        END
      ELSE { y2 >= 0 i.e. y2 >= 1 }
        IF IsEven(y2) THEN
          IF x2 < 0.0 THEN BEGIN
            IF test THEN writeln('Case 3');
            Pwr:= p(x2,y1) +* p(x1,y2);
          END
          ELSE { x2 >= 0 } BEGIN
            IF test THEN writeln('Case 4');
            Pwr:= intval(0.0) +* p(MaxAbs(x1, x2), y2);
          END
        ELSE { y2 is an odd number } BEGIN
          IF test THEN writeln('Case 5');
          Pwr:= p(x1,y1) +* p(x2,y2);
        END
    END
    ELSE { x1 < 0 and (y not a point or y not an integer value) }
    BEGIN
      { Case: Err2; inf(x) < 0 and  not (y integer <> 0) }
      Pwr:= error(2);
    END;
  END { cases x1 < 0 }
  ELSE  { x1 >= 0.0 }
    IF x1 = 0 THEN
      IF y1 < 0 THEN BEGIN
        { Case: Err3; 0 in x and inf(y) < 0 }
        Pwr:= error(3)
      END
      ELSE { y1 >= 0.0 (==> y2 > 0)}
        IF x2 <= 1.0 THEN
          IF y1 = 0 THEN BEGIN
            { For x = 0 and inf(y) = 0 the result 0**[0, sup(y)]   }
            { consists out of the two discrete values 0 and 1.     }
            { E.g. 0**[0,sup(y)] = [0, 1]                          }
            { [0, sup(x)>0]**[0, sup(y)>0] = [0, 1] }
            IF test THEN writeln('Case 6');
            Pwr:= intval(0.0, 1.0)
          END
          ELSE { y1 > 0.0 } BEGIN
            IF test THEN writeln('Case 7');
            Pwr:= intval(0.0) +* p(x2,y1);
          END
        ELSE BEGIN { x2 > 1.0 }
          IF test THEN writeln('Case 8');
          Pwr:= intval(0.0) +* p(x2,y2);
        END
    ELSE { x1 > 0.0 }
      IF x2 <= 1.0 THEN
        IF y2 <= 0.0 THEN BEGIN
          IF test THEN writeln('Case 9');
          Pwr:= p(x2,y2) +* p(x1,y1);
        END
        ELSE
          IF y1 >= 0.0 THEN BEGIN
            IF test THEN writeln('Case 10');
            Pwr:= p(x1,y2) +* p(x2,y1);
          END
          ELSE { y1 < 0.0 < y2 } BEGIN
            IF test THEN writeln('Case 11');
            Pwr:= p(x1,y2) +* p(x1,y1);
          END
      ELSE
```

```
          IF x1 >= 1.0 THEN
            IF y2 <= 0.0 THEN BEGIN
              IF test THEN writeln('Case 12');
              Pwr:= p(x2,y1) +* p(x1,y2);
            END
            ELSE
              IF y1 >= 0.0 THEN BEGIN
                IF test THEN writeln('Case 13');
                Pwr:= p(x1,y1) +* p(x2,y2);
              END
              ELSE { y1 < 0.0 < y2 } BEGIN
                IF test THEN writeln('Case 14');
                Pwr:= p(x2,y1) +* p(x2,y2);
              END
          ELSE { x1 < 1.0 < x2 }
            IF y2 <= 0.0 THEN BEGIN
              IF test THEN writeln('Case 15');
              Pwr:= p(x2,y1) +* p(x1,y1);
            END
            ELSE
              IF y1 >= 0.0 THEN BEGIN
                IF test THEN writeln('Case 16');
                Pwr:= p(x1,y2) +* p(x2,y2);
              END
              ELSE { y1 < 0.0 < y2 } BEGIN
                IF test THEN writeln('Case 17');
                Pwr:= p(x1,y2) +* p(x2,y1)
                      +* p(x2,y2) +* p(x1,y1);
              END;
999:
END; { function Pwr }
{-----------------------------------------------------------------}

BEGIN
  test:= false;    { global variable; default setting: no messages }
END.
{-----------------------------------------------------------------}
```

9.8.5 Test Cases for the Power Function

A complete set of test cases (some cases appear more than once) is supported by the following arguments. The data are listed in the form
Line number x-interval, y-interval.

```
1         0             0
2         2             0
3         [-7, 2]       0
4         [-2, 1]       0
5         -5            -2
6         [-2, -1]      -2
7         -5            -3
8         [-2, -1]      -1
9         -5            2
10        [-2, -1]      2
11        [-2, 0]       2
12        [-2, 3]       2
13        -5            3
14        [-2, -1]      3
15        [-2, 0]       3
```

16	[-2, 1]	3
17	0	[0, 0.5]
18	0	[0, 2]
19	[0, 1]	[0, 0.5]
20	[0, 0.5]	[0, 0.5]
21	[0, 0.5]	[1, 2]
22	0	2
23	[0, 1]	0.5
24	[0, 4]	[0.5, 1]
25	[0, 4]	0.5
26	[0, 4]	[0, 0.5]
27	[0.5, 1]	[-1, 0]
28	[0.25, 1]	[-1, -0.5]
29	1	1
30	[0.5, 1]	[0, 1]
31	0.5	[1, 2]
32	[0.5, 1]	[-1, 1]
33	[1, 2]	[-2, 0]
34	[1, 2]	[-2, -1]
35	[2, 4]	[-2, -1]
36	[1, 2]	[0, 1]
37	[1, 2]	[1, 2]
38	[4, 5]	[0.5, 2]
39	[1, 4]	[-1, 1]
40	[0.25, 4]	[-2, 0]
41	[0.25, 4]	[-2, -1]
42	[0.25, 4]	[0, 0.5]
43	[0.25, 4]	[0.5, 2]
44	[0.25, 5]	[-2, 2]
45	[0.25, 4]	[-2, 2]
46	[-1, 1]	-1
47	[-2, 1]	0.5
48	0	-1
49	[0, 1]	-1

These data are handeled by the program **pwrgen** to generate a set of reference values that may be used to perform a consistency check on other machines.

```
{---------------------------------------------------------------}
{ The reference file pwrtst.ref will be  generated.             }
{ The reference values will be used by the program              }
{ pwrtst.p to perform a consistency check for the               }
{ function Pwr (interval power function) in module pwrm.        }
{---------------------------------------------------------------}
PROGRAM pwrgen;

{ This program uses the data file pwrtst.dat as its input file.    }

{ The file pwrtst.ref will be generated. An existing }
{ file with this name will be overwritten!           }

USE i_ari,  { Interval operations                        }
  pwrm;     { Make available the interval power function Pwr }

VAR
  RefVal, Inp: text;
  x, y, res: interval;
  i: integer;

BEGIN
  writeln('Generation of reference values for the program pwrtst.p');
  reset(Inp, 'pwrtst.dat');     { Data values               }
  writeln('File to store the reference values (pwrtst.ref)?  ');
```

```
  rewrite(RefVal);              { Generated reference values }
  test:= true; { Enable printing of error messages in module pwr.p }
  WHILE NOT eof(Inp) DO BEGIN
    readln(Inp, i, x, y);  { Read data }
    res:= Pwr(x, y);
    writeln(i:3, '  ',  res.inf:20:13:-1, '  ', res.sup:20:13:+1);
    IF res.inf > res.sup THEN writeln('***** E R R O R  in pwrgen!!!!');
    writeln(RefVal,
            i:3, '  ',  res.inf:20:13:-1, '  ', res.sup:20:13:+1);
  END;
END.
```

Here are the first lines of the generated file `pwrgen.ref`.

```
  1       1.0000000000000      1.0000000000000
  2       1.0000000000000      1.0000000000000
  3       1.0000000000000      1.0000000000000
  4       1.0000000000000      1.0000000000000
  5       0.0399999999999      0.0400000000001
  6       0.2499999999999      1.0000000000000
  7      -0.0080000000001     -0.0079999999999
  8      -1.0000000000000     -0.4999999999999
  9      24.9999999999999     25.0000000000001
 10       1.0000000000000      4.0000000000001
 11       0.0000000000000      4.0000000000001
 12       0.0000000000000      9.0000000000001
 13    -125.0000000000001   -124.9999999999999
```

• • •

For each argument the output shows the resulting interval as well as the case number. This number refers to the column indicated by *case* of Table 9.2.
The program `pwrtst` listed now uses these reference values stored in the file `pwrgen.ref` to perform a consistency check.

```
{-----------------------------------------------------------------------}
{               Check Interval Power Function                          }
{                                                                       }
{ Performs a consistency check for the function pwr in module pwrm.p   }
{ The files pwrtst.dat and pwrtst.ref must be available!               }
{ The reference file pwrtst.ref has been generated using               }
{ the program pwrgen.p. This program uses the same data                }
{ file pwrtst.dat.                                                      }
{-----------------------------------------------------------------------}
PROGRAM pwrtst;
USE i_ari,  { Interval operations }
pwrm;       { Make available the interval power function pwr }

VAR
  data, reference, new: text;
  x, y, res: interval;
  line1, line2: string;
  i: integer;
  ErrorFlag: boolean;
BEGIN
  test:= true;       { Monitioring of test cases. }
  { test is a global variable of the module pwrm. }
  { test = true gives additional information and  }
  { alters the error handling.                    }

  reset(data,  'pwrtst.dat');
  rewrite(new, 'pwrtst.tmp');
  WHILE NOT eof(data) DO BEGIN
```

```
    readln(data, i, x, y);
    res:= pwr(x, y);
    writeln(     i:3, '  ',  res.inf:20:13:-1, '  ', res.sup:20:13:+1);
    writeln(new, i:3, '  ',  res.inf:20:13:-1, '  ', res.sup:20:13:+1);
  END;

  { Check whether the generated output is correct.              }
  { The file pwrtst.ref has been generated on a SUN workstation. }
  reset(reference, 'pwrtst.ref');
  reset(new);
  i:= 1;
  ErrorFlag:= false; { Initialize ErrorFlag (no error) }
  WHILE NOT eof(new) DO BEGIN  { Check line by line just generated }
    readln(reference, line1); {   results with reference values   }
    readln(new, line2);       {   coming from file pwrtst.ref     }
    IF line1 <> line2 THEN ErrorFlag:= true;
    i:= i + 1;
  END;
  writeln;
  IF ErrorFlag THEN
    writeln('ERROR in pwrgen: Not all results are consistent!')
  ELSE
    writeln('pwrgen: Results are all consistent!');
END.
{-----------------------------------------------------------------}
```

The driver program `pwrtst` produces the following output (only the last few lines are shown):

• • •

```
 43       0.0624999999999      16.0000000000001
Case 17
 44       0.0399999999999      25.0000000000001
Case 17
 45       0.0624999999999      16.0000000000001
Error code: 1
****** ERROR:  Invalid arguments for pwr !!!!!
 46       1.0000000000000       1.0000000000000
Error code: 2
****** ERROR:  Invalid arguments for pwr !!!!!
 47       2.0000000000000       2.0000000000000
Error code: 3
****** ERROR:  Invalid arguments for pwr !!!!!
 48       3.0000000000000       3.0000000000000
Error code: 3
****** ERROR:  Invalid arguments for pwr !!!!!
 49       3.0000000000000       3.0000000000000

pwrgen: Results are all consistent!
```

The lines ranging from 45 (exclusively) to 49 are caused by invalid arguments. Compare the PASCAL-XSC listing of the module `pwrm.p`. The most important line of the output is the last one.

The values included in file `pwrgen.ref` can be used together with the driver program `pwrtst` to check whether an installation of PASCAL-XSC on an different machines or using a different C compiler on the same machine produces identical results. If the results are not identical with the supplied reference values the installation has to be checked more thoroughly.

Chapter 10

Verified Integration

In this Chapter we will discuss some simple methods and programs for the verified computation of definite integrals. Many widely used classes of numerical integration formulas for the determination of the integral

$$I = \int_a^b f(x)dx \tag{10.1}$$

are of the form

$$I = J + R \tag{10.2}$$

where the integral I is approximated by the approximation term J which is the scalar product of certain weights w_i and the function values of f at certain nodes x_i:

$$J = \sum_{i=0}^{n} w_i f(x_i) \ . \tag{10.3}$$

The second term R in (10.2) is the remainder term of the integration formula. For sufficiently smooth integrands f this remainder term can often be expressed in a form containing higher derivatives of f at some unknown intermediate points of the interval of integration $[a, b]$.
Integration rules of this type are e.g. the Newton-Côtes formulas, the well known Romberg integration or Gaussian quadrature where in the latter case an additional weight function $w(x)$ appears in the integrand of (10.1).
The basic principle for verified computation of I is the same for all these formulas: we compute enclosures for the function values $f(x_i)$ and compute the scalar product J in interval arithmetic. For the remainder term R we use automatic differentiation (see Chapter 5.1) to compute the higher derivatives which appear in the expression for R. The unknown intermediate points are replaced by the interval $[a, b]$ or by suitable subintervals (e.g. $[x_i, x_{i+1}]$) which guarantees that these unknown quantities are enclosed properly.
If an absolute error bound is given which has to be satisfied, we can proceed in the following way: starting from $h_0 = b - a$ we compute for decreasing step sizes h_i enclosures for the remainder term R until its diameter is less than the given absolute error bound. Then we compute an enclosure of the approximation term J for the last step size and add it to the enclosure of the remainder term which results in an enclosure of the integral value. If, however, a relative error bound is given then in

general it is not sufficient to look at the remainder term only in order to decide if the required accuracy is satisfied. Here, in addition, we have to take into account an approximation J for the integral (which needs to be correct only in the order of magnitude, however).
For simplicity we confine ourselves to the widely used Trapezoidal and Simpson's integration rules for which we can follow exactly the pattern outlined above and to Romberg integration, where the approximation term can be computed as scalar product (10.3) or by the well known recursive extrapolation algorithm.

10.1 Trapezoidal and Simpson's Rule

10.1.1 Theoretical Background

Both, the trapezoidal as well as Simpson's rule are Newton-Côtes formulas. For sufficently smooth functions f the trapezoidal rule

$$I = \frac{b-a}{2}(f(a)+f(b)) \quad - \quad \frac{(b-a)^3}{12}f^{(2)}(\xi), \quad \xi \in [a,b] \tag{10.4}$$

and Simpson's rule

$$I = \frac{b-a}{6}(f(a)+4f(\frac{a+b}{2})+f(b)) \quad - \quad \frac{(b-a)^5}{90}f^{(4)}(\xi), \quad \xi \in [a,b] \tag{10.5}$$

are usually used in the iterated form. This means that we subdivide the interval $[a,b]$ into n equidistant subintervals of length $h=(b-a)/n$ with the nodes $x_i = a+ih, i=0\dots n$. In the case of Simpson's rule n must be even. Now we apply the integration rule on each subinterval separately (on each pair of subintervals for Simpson's rule) and accumulate the approximating terms and the remainder terms. For the trapezoidal rule we then get from n subintervals of length h each

$$I = J_T + R_T \tag{10.6}$$

with the approximating term

$$J_T = \frac{h}{2}\sum_{i=0}^{n}{}^{T}\, 2f(x_i) = \frac{h}{2}(f(x_0)+2f(x_1)+2f(x_2)+\cdots+2f(x_{n-1})+f(x_n)) \tag{10.7}$$

and the remainder term

$$R_T = -\frac{h^3}{12}\sum_{i=1}^{n} f^{(2)}(\xi_i) \tag{10.8}$$

where the first sum is marked with a "T" to indicate that the first and the last term in the summation (i.e. $f(x_0)$ and $f(x_n)$) must be multiplied with a factor of 1 only instead of 2. The second derivative of f is evaluated at unknown intermediate points $\xi_i \in [x_{i-1}, x_i]$ in the second sum.

Similarly, for Simpson's rule we obtain from $n/2$ subintervals of length $2h$ each

$$I = J_S + R_S \tag{10.9}$$

with the approximating term

$$J_S = \frac{h}{3} \sum_{i=0}^{n} s_i f(x_i) \tag{10.10}$$

and the remainder term

$$R_S = -\frac{h^5}{90} \sum_{i=1}^{n/2} f^{(4)}(\xi_i) \tag{10.11}$$

where $s_i = 4$ for i odd and $s_i = 2$ for i even except for $i = 0$ and $i = n$ where $s_0 = s_n = 1$. Again the derivatives in the second sum are evaluated at intermediate points $\xi_i \in [x_{2i-2}, x_{2i}]$ which are in general different from those in (10.8), of course. Now the computation of enclosures for I with either rule is straight forward. We have to compute enclosures of J_T and R_T or for J_S and R_S.
To enclose the approximation terms we have to compute enclosures $\Diamond f(x_i)$ of the function values of f at the nodes i.e. we have to evaluate $f(x_i)$ in interval arithmetic. Also J_T and J_S are then evaluated in interval arithmetic:

$$J_T \in [J_T] := h \mathbin{\Diamond\!\!\!\!/} 2 \mathbin{\Diamond\!\!\!\!\ast} \Diamond\sum_{i=0}^{n}{}^{T}\; 2 \mathbin{\Diamond\!\!\!\!\ast} \Diamond f(x_i) \tag{10.12}$$

and

$$J_S \in [J_S] := h \mathbin{\Diamond\!\!\!\!/} 3 \mathbin{\Diamond\!\!\!\!\ast} \Diamond\sum_{i=0}^{n} s_i \mathbin{\Diamond\!\!\!\!\ast} \Diamond f(x_i)\ . \tag{10.13}$$

We can even gain more accuracy if we accumulate the sums in (10.12) and (10.13) by use of a long accumulator i.e. by computing the sums in (10.12) and (10.13) exactly and rounding them only once to a floating-point interval:

$$J_T \in [J_T] := h \mathbin{\Diamond\!\!\!\!/} 2 \mathbin{\Diamond\!\!\!\!\ast} \Diamond\sum_{i=0}^{n}{}^{T}\; 2 \Diamond f(x_i) \tag{10.14}$$

and

$$J_S \in [J_S] := h \mathbin{\Diamond\!\!\!\!/} 3 \mathbin{\Diamond\!\!\!\!\ast} \Diamond\sum_{i=0}^{n} s_i \Diamond f(x_i)\ . \tag{10.15}$$

Enclosures for the remainder terms will be computed by replacing the intermediate points ξ_i by enclosing intervals, i.e. by $[x_{i-1}, x_i]$ for the trapezoidal rule and by $[x_{2i-2}, x_{2i}]$ for Simpson's rule. Evaluating the derivatives of f over these intervals we obtain the enclosures

$$R_T \in [R_T] := -h^3 \mathbin{\Diamond\!\!\!\!/} 12 \mathbin{\Diamond\!\!\!\!\ast} \Diamond \sum_{i=1}^{n} \Diamond f^{(2)}([x_{i-1}, x_i]) \tag{10.16}$$

and

$$R_S \in [R_S] := -h^5 \mathbin{\Diamond\!\!\!/} 90 \mathbin{\Diamond\!\!\!\ast} \Diamond \sum_{i=1}^{n/2} \Diamond f^{(4)}([x_{2i-2}, x_{2i}]) \tag{10.17}$$

Finally the integral I is contained in $[J_T] + [R_T]$ for the trapezoidal rule and in $[J_S] + [R_S]$ for Simpson's rule.
The diameter of $[R_T]$ can easily be estimated by $c_T h^2$ and the diameter of $[R_S]$ by $c_S h^4$ where the constants c_T and c_S do not depend on the step size h. Therefore, since the diameter of $[J_T]$ and that of $[J_S]$ are independent of h, the diameter of the enclosure of I also decreases with second order in h for the trapezoidal and with fourth order for Simpson's rule.

10.1.2 Algorithms

Here we list the verifying algorithms for the trapezoidal rule and for Simpson's rule in Pseudo-Pascal notation.
The termination criterion of the algorithms is chosen as follows: We start with one or two subintervals, compute an enclosure of the remainder term and double the number of subintervals if the diameter of the computed remainder term is not smaller than a user supplied absolute error bound abs_err. We repeat this until the error bound is satisfied or until the diameter of the remainder term starts to increase again. Finally we compute an enclosure of the approximation term and add it to the remainder term to obtain an enclosure of the integral.

Algorithm 10.1: Integral $(f, a, b, abs\_err, error)$ {function}

{Trapezoidal Rule with Verification}
input: function f, integration bounds a, b error bound abs_err
output: error indicator, function value (enclosure of integral value)

(a) determine number n of subintervals :
  $n := 1$
  $h := (b-a)/n$
  $R := -h^3/12 \cdot \Diamond f^{(2)}([a,b])$
  repeat
    $n := 2n$
    $h := (b-a)/n$
    $R_{old} := R$
    $R := -h^3/12 \cdot \Diamond \sum_{i=1}^{n} \Diamond f^{(2)}([x_{i-1}, x_i])$
  until $\mathrm{diam}(R) \le abs\_err$ **or** $\mathrm{diam}(R) > \mathrm{diam}(R_{old})$

(b) compute enclosure of integral :
  Integral $:= R + h/2 \cdot \Diamond \sum_{i=0}^{n}{}^{T} 2 \Diamond f(x_i)$

(c) determine accuracy :
if diam(Integral) $\leq abs\_err$ **then** $error := 0$ {accuracy sufficient}
else $error := 1$ {accuracy insufficient}

(d) **return** Integral, $error$

Algorithm 10.2: Integral $(f, a, b, abs\_err, error)$ {function}

{Simpson's Rule with Verification}
input: function f, integration bounds a, b, error bound abs_err
output: error indicator, function value (enclosure of integral value)

(a) determine number n of subintervals :
$n := 2$
$h := (b - a)/n$
$R := -h^5/90 \cdot \Diamond f^{(4)}([a, b])$
repeat
$n := 2n$
$h := (b - a)/n$
$R_{old} := R$
$R := h^5/90 \cdot \Diamond \sum_{i=1}^{n/2} \Diamond f^{(4)}([x_{2i-2}, x_{2i}])$
until $\text{diam}(R) \leq abs\_err$ **or** $\text{diam}(R) > \text{diam}(R_{old})$

(b) compute enclosure of integral :
Integral $:= R + h/3 \cdot \Diamond \sum_{i=0}^{n} s_i \Diamond f(x_i)$

(c) determine accuracy :
if diam(Integral) $\leq abs\_err$ **then** $error := 0$ {accuracy sufficient}
else $error := 1$ {accuracy insufficient}

(d) **return** Integral, $error$

10.1.3 Applicability of the Algorithms

As already discussed in Section 10.1.1 the algorithms may be applied to any function f which is sufficiently smooth. More precisely, since we compute the derivatives of f by automatic differentiation, we can treat sufficiently smooth functions which are composed of operations and functions from the module itaylor.
Also functions which are nonsmooth at finitely many points in $[a, b]$ may be treated with these algorithms if $[a, b]$ is subdivided such that each of these points is a boundary point of some subinterval and the algorithm is applied in these subintervals.
In certain cases the user required accuracy abs_err may not be satisfied after completion of the algorithms. This will primarily be the case if the accuracy demand is very high compared to the overestimations which occur in the interval evaluations of f in the approximation term. This may happen especially if the value of the integral is close to zero or if f is a rather complicated expression.

Alternatively a relative error criterion could be used, however, the overestimations will still remain and limit the reachable accuracy. Possible ways out of this situation are the use of higher precision, integration formulas of higher order and adaptive strategies.

10.1.4 PASCAL-XSC Program Code

The Algorithms are coded as a global function `INTEGRAL` in a module `itrapez` and a module `isimpson`. `INTEGRAL` takes as input parameters the function f to be integrated as PASCAL-XSC function, the integration bounds `a,b` and the required absolute error `abs_err`. As the functions result it returns an interval enclosing the value of the integral. The parameter `error` returns 0 if the required accuracy could be met otherwise a value of 1 is returned.
Here we remind that the derivatives which are computed in the module itaylor are really Taylor-coefficients, i.e. they are the derivatives of f divided by the corresponding factorials. Therefore the constants used for the computation of the remainder term differ between the program codes and the algorithms.
First we list the module `itrapez` which uses the Trapezoidal rule:

```
MODULE itrapez;

USE itaylor;

{-----------------------------------------------------------------------}
{                                                                       }
{ The function integral contained in itrapez computes an enclosure for  }
{ the value of the definite integral of a function f(x) over the interval}
{ [a,b] by use of the iterated trapezoidal rule.                        }
{                                                                       }
{ itrapez uses the operators and functions of the module itaylor for the }
{ computation of the second derivative of f in the remainder term by use }
{ of automatic differentiation.                                         }
{                                                                       }
{ The parameters of the global function integral are                    }
{   f       : function to be integrated, argument and result as itaylor  }
{   a,b     : integration bounds                                        }
{   abs_err : upper bound for the absolute error required from user     }
{   error   : error indicator upon return of function  integral         }
{             error = 0  result meets required accuracy                 }
{             error = 1  result has diameter > abs_err                  }
{                                                                       }
{                                                  b                    }
{   function's result : interval enclosing  integral  f(x) dx           }
{                                           a                           }
{-----------------------------------------------------------------------}


GLOBAL FUNCTION INTEGRAL( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                                   a,b     : REAL;
                                   abs_err : REAL;
                               VAR error   : INTEGER )  : INTERVAL;

VAR i,n             : INTEGER;
    R,R_old,
    integral_value : INTERVAL;
```

```
FUNCTION remainder( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                    n : INTEGER ) : INTERVAL;
{-----------------------------------------------------------------------}
{ Computes for each sub-interval of [a,b] an enclosure of the remainder  }
{ term of the trapezoidal rule and accumulates these enclosures. The     }
{ resulting interval is an enclosure of the remainder term on the whole  }
{ interval [a,b].                                                        }
{ The accumulation could be done in dotprecision variables, however,     }
{ the roundoff errors of this accumulation do not affect the integral    }
{ enclosure as much as the roundoff errors of the approximation part.    }
{-----------------------------------------------------------------------}
VAR x,fx : itaylor[0..2];
    s,h  : INTERVAL;
    i    : INTEGER;
BEGIN
   h:= ( INTVAL(b) - INTVAL(a) ) / n;
   s:= 0.0;
   FOR i:= 0 TO n-1 DO
   BEGIN
      expand( x, INTVAL(a) + h*INTVAL(i,i+1) );
      fx:= f( x );
      s:= s + fx[2];
   END;
   remainder:= h*sqr(h)*s/6;
END;

FUNCTION trapezoidal( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                      n : INTEGER ) : INTERVAL;
{-----------------------------------------------------------------------}
{ Computes the iterated trapezoidal rule on the interval [a,b]. The      }
{ function values of f are computed in interval arithmetic to take all   }
{ roundoff errors into account. To avoid unnecessary additional roundoff }
{ errors the accumulation is done in two dotprecision variables lo,up    }
{ ( for the inf and the sup of the sum ).                                }
{-----------------------------------------------------------------------}
VAR x,fx  : itaylor[0..0];
    h     : INTERVAL;
    lo,up : DOTPRECISION;
    i     : INTEGER;
BEGIN
   h:= ( INTVAL(b) - INTVAL(a) ) / n;
   lo:= #(0.0);
   up:= #(0.0);
   FOR i:= 0 TO n DO
   BEGIN
      expand( x, INTVAL(a) + i*h );
      fx:= f( x );
      IF (i=0) OR (i=n) THEN BEGIN
                                lo:= #( lo + 0.5*inf(fx[0]) );
                                up:= #( up + 0.5*sup(fx[0]) );
                             END
                        ELSE BEGIN
                                lo:= #( lo + inf(fx[0]) );
                                up:= #( up + sup(fx[0]) );
                             END;
   END;
   trapezoidal:= h * intval( #<(lo), #>(up) );
END;

BEGIN  { INTEGRAL }
   n:= 1;
   R:= remainder( f, n );
   { simple strategy : always double number of sub-intervals until }
   { diam(remainder) is less than abs_err or starts to grow again  }
```

```
   REPEAT
      n    := 2*n;
      R_old:= R;
      R    := remainder( f, n );
   UNTIL      ( DIAM(R) <= abs_err )       { remainder small enough }
         OR  ( DIAM(R) >= DIAM(R_old) );  { remainder grows again  }

   { integral = approximation + remainder }
   integral_value:= trapezoidal( f, n ) - R;

   IF diam(integral_value) <= abs_err THEN error:= 0
                                      ELSE error:= 1;

   INTEGRAL:= integral_value;
END;   { INTEGRAL }

END.
```

Next we list the module simpson which uses Simpson's rule:

```
MODULE isimpson;

USE itaylor;

{-----------------------------------------------------------------------}
{                                                                       }
{ The function integral contained in isimpson computes an enclosure for  }
{ the value of the definite integral of a function f(x) over the interval}
{ [a,b] by use of the iterated Simpson's rule.                          }
{                                                                       }
{ isimpson uses the operators and functions of the module itaylor for the}
{ computation of the fourth derivative of f in the remainder term by use }
{ of automatic differentiation.                                         }
{                                                                       }
{ The parameters of the global function integral are                    }
{   f       : function to be integrated, argument and result as itaylor  }
{   a,b     : integration bounds                                        }
{   abs_err : upper bound for the absolute error required from user      }
{   error   : error indicator upon return of function  integral          }
{             error = 0  result meets required accuracy                  }
{             error = 1  result has diameter > abs_err                   }
{                                                                       }
{                                                    b                  }
{   function's result : interval enclosing  integral  f(x) dx            }
{                                           a                           }
{-----------------------------------------------------------------------}


GLOBAL FUNCTION INTEGRAL( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                                   a,b     : REAL;
                                   abs_err : REAL;
                               VAR error   : INTEGER )  : INTERVAL;

VAR j,n            : INTEGER;
    R,R_old,
    integral_value : INTERVAL;

FUNCTION remainder( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                    n : INTEGER ) : INTERVAL;
{-----------------------------------------------------------------------}
{ Computes for each sub-interval of [a,b] an enclosure of the remainder  }
```

```
{ term of Simpson's rule and accumulates these enclosures. The            }
{ resulting interval is an enclosure of the remainder term on the whole  }
{ interval [a,b].                                                         }
{ The accumulation could be done in dotprecision variables, however,     }
{ the roundoff errors of this accumulation do not affect the integral    }
{ enclosure as much as the roundoff errors of the approximation part.    }
{------------------------------------------------------------------------}
VAR x,fx : itaylor[0..4];
    s,h  : INTERVAL;
    i,j  : INTEGER;
BEGIN
   h:= ( INTVAL(b) - INTVAL(a) ) / n;
   s:= 0.0;
   FOR i:= 1 TO n DIV 2 DO
   BEGIN
      j:= 2*(i-1);
      expand( x, INTVAL(a) + h*INTVAL(j,j+2) );
      fx:= f( x );
      s:= s + fx[4];
   END;
   remainder:= 4*h*sqr(sqr(h))*s/15;
END;

FUNCTION simpson( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                  n : INTEGER ) : INTERVAL;
{------------------------------------------------------------------------}
{ Computes the iterated Simpson rule on the interval [a,b]. The          }
{ function values of f are computed in interval arithmetic to take all   }
{ roundoff errors into account. To avoid unnecessary additional roundoff }
{ errors the accumulation is done in two dotprecision variables lo,up    }
{ ( for the inf and the sup of the sum ).                                }
{------------------------------------------------------------------------}
VAR x,fx  : itaylor[0..0];
    h     : INTERVAL;
    lo,up : DOTPRECISION;
    j     : INTEGER;
BEGIN
   h:= ( INTVAL(b) - INTVAL(a) ) / n;
   lo:= #(0.0);
   up:= #(0.0);
   FOR j:= 0 TO n DO
   BEGIN
      expand( x, INTVAL(a) + j*h );
      fx:= f( x );
      IF (j=0) OR (j=n) THEN BEGIN
                                lo:= #( lo + inf(fx[0]) );
                                up:= #( up + sup(fx[0]) );
                             END
                        ELSE
      IF  j MOD 2 = 0   THEN BEGIN
                                lo:= #( lo + 2.0*inf(fx[0]) );
                                up:= #( up + 2.0*sup(fx[0]) );
                             END
                        ELSE BEGIN
                                lo:= #( lo + 4.0*inf(fx[0]) );
                                up:= #( up + 4.0*sup(fx[0]) );
                             END;
   END;
   simpson:= h * intval( #<(lo),#>(up) ) / 3.0;
END;

BEGIN  { INTEGRAL }
   n:= 2;
   R:= remainder( f, n );
```

```
   { simple strategy : always double number of sub-intervals until }
   { diam(remainder) is less than abs_err or starts to grow again  }
   REPEAT
      n    := 2*n;
      R_old:= R;
      R    := remainder( f, n );
   UNTIL     ( DIAM(R) <= abs_err )       { remainder small enough }
         OR  ( DIAM(R) >= DIAM(R_old) );  { remainder grows again  }

   { integral = approximation + remainder }
   integral_value:= simpson( f, n ) - R;

   IF diam(integral_value) <= abs_err THEN error:= 0
                                      ELSE error:= 1;

   INTEGRAL:= integral_value;
END;   { INTEGRAL }

END.
```

10.1.5 Test Results

We consider a simple integral which conains a parameter c.

$$I(c) := \int_{-1}^{1} \frac{c^2}{1+(cx)^2} dx \tag{10.18}$$

The exact value of the integral is $I(c) = 2c^{-1}\arctan(c^{-1})$. For large values of c the integrand $f(x) = c^2/(1+(cx)^2)$ exhibits a sharp peak at $x = 0$ which makes the integration difficult for such simple algorithms as our equidistant trapezoidal and Simpson's rules. This can be seen very clearly for the two cases $c = 1$ and $c = 10$. The following short PASCAL-XSC program has been used to compute the results contained in the tables in this section (the program printed here computes the case $c = 10$). For the application of Simpson's rule the use-clause in the program has to be replaced by `USE isimpson;`)

```
PROGRAM example;

USE i_ari,itaylor;
USE itrapez;


FUNCTION f ( x : itaylor ) : itaylor[lb(x)..ub(x)];
BEGIN
   f := 100 / ( 1 + sqr(10*x) );
END;

VAR a,b,abs_err : REAL;
    error       : INTEGER;
    int         : INTERVAL;

BEGIN
   write( 'a,b = ' ); read(a,b);
   REPEAT
      write( 'abs_err = ' ); read( abs_err );
      IF abs_err > 0 THEN
```

```
      BEGIN
         int:= integral( f, a,b, abs_err, error );
         writeln( int, '  ', diam(int):15, error:3 );
      END;
   UNTIL abs_err <= 0;
END.
```

abs_err	$[I(1)]$	$d([I(1)])$	n	error
1E+00	$1.{}_{4}^{7}$	1.37E-01	4	0
1E-01	1.5_{6}^{8}	1.51E-02	8	0
1E-02	1.5_{69}^{72}	1.81E-03	16	0
1E-04	1.570_{7}^{9}	2.78E-05	64	0
1E-06	1.57079_{6}^{7}	4.34E-07	256	0
1E-08	1.5707963_{2}^{4}	6.78E-09	1024	0
1E-10	1.570796326_{78}^{81}	1.33E-11	8192	0
1E-12	1.57079632679_{4}^{6}	2.08E-13	32768	0
1E-14	1.570796326794_{89}^{90}	3.78E-15	131072	0
1E-15	1.57079632679489_{6}^{8}	1.12E-15	262144	1

Table 10.1: Results for I(1) with trapezoidal rule

abs_err	$[I(10)]$	$d([I(10)])$	n	error
1E+00	$[29, 30]$	4.66E-01	64	0
1E-01	$29.{}_{3}^{5}$	5.50E-02	128	0
1E-02	29.4_{1}^{3}	6.76E-03	256	0
1E-04	29.4225_{4}^{6}	1.32E-05	2048	0
1E-06	29.422553_{3}^{6}	2.06E-07	8192	0
1E-08	29.42255348_{8}^{9}	3.21E-09	32768	0
1E-10	29.422553486_{0}^{1}	5.01E-11	131072	0
1E-12	29.42255348607_{4}^{6}	7.93E-13	524288	0
1E-14	$29.422553486074_{68}^{71}$	1.78E-14	4194304	1

Table 10.2: Results for I(10) with trapezoidal rule

For increasing demands of accuracy the trapezoidal rule needs excessively many nodes (Tables 10.1 and 10.2) whereas Simpson's rule gets the same accuracy with considerably less nodes reflecting the higher order of Simpson's rule. Especially the case $c = 10$ demonstrates that for an efficient integration algorithm the constant step size must be abandoned and be replaced by a suitable step size adaption. We will revisit this case later in Section 10.2.6 where an adaptive algorithm will be applied with much higher efficiency.
In the tables 10.1 – 10.4 the first column contains the user supplied absolute error bound `abs_err`, in the second column the computed enclosures $[I(c)]$ are listed

whose diameters $d([I(c)])$ can be found in the third column. The fourth column contains the number n of nodes (i.e. function evaluations of f) which were used in the integration. The last column is the error indicator of the integration routine which shows if the required accuracy could be satisfied (`error` = 0) or not (`error` = 1).

`abs_err`	$[I(1)]$	$d([I(1)])$	n	`error`
1E+00	$1.^{8}_{3}$	4.42E-01	4	0
1E-01	1.5^{8}_{6}	6.12E-03	8	0
1E-02	1.5^{8}_{6}	6.12E-03	8	0
1E-04	1.570^{80}_{79}	3.38E-06	32	0
1E-06	1.570796^{4}_{2}	1.02E-07	64	0
1E-08	1.5707963^{3}_{2}	3.14E-09	128	0
1E-10	1.570796326^{9}_{7}	9.79E-11	256	0
1E-12	1.57079632679^{50}_{48}	9.66E-14	1024	0
1E-14	1.570796326794^{90}_{89}	4.00E-15	2048	0
1E-15	1.57079632679489^{8}_{5}	1.34E-15	4096	1

Table 10.3: Results for I(1) with Simpson's rule

`abs_err`	$[I(10)]$	$d([I(10)])$	n	`error`
1E+00	$[29, 30]$	3.17E-01	64	0
1E-01	29.4^{3}_{1}	5.86E-03	128	0
1E-02	29.4^{3}_{1}	5.86E-03	128	0
1E-04	29.4225^{6}_{5}	4.25E-06	512	0
1E-06	29.422553^{6}_{4}	1.31E-07	1024	0
1E-08	29.4225534^{9}_{8}	4.06E-09	2048	0
1E-10	29.4225534860^{8}_{7}	3.98E-12	8192	0
1E-12	29.422553486074^{8}_{6}	1.43E-13	16384	0
1E-14	$29.422553486074^{71}_{67}$	2.14E-14	32768	1

Table 10.4: Results for I(10) with Simpson's rule

10.2 Romberg-Integration

10.2.1 Theoretical Background

Romberg integration is a numerical integration method for sufficiently smooth functions which is based on the trapezoidal rule evaluated for different step sizes and supplemented by a suitable extrapolation procedure. It can also be written in the form (10.2) where the remainder term R contains a higher derivative of f and the approximation term J is of the form of a scalar product (10.3).

Usually, however, the weights occurring in J are not used explicitly, but rather J is computed recursively from a so-called extrapolation table. We now give a short description of this recursive computation. For more details see e.g. [309], [94] or [97].

For a sequence of decreasing step sizes $h_0 > \ldots > h_m$ we compute the trapezoidal sums as defined in equation (10.7) which we denote as $T(h_i),\ i = 0, \ldots, m$, here. Then we consider the interpolation polynomials

$$\tilde{T}_{ik}(h) = a_{ik}^{(0)} + a_{ik}^{(1)} h^2 + \ldots + a_{ik}^{(m)} h^{2k}$$

in h^2 which interpolates the $k+1$ values $T(h_j),\ j = i-k, \ldots, i$, i.e.

$$\tilde{T}_{ik}(h_j) = T(h_j),\ j = i-k, \ldots, i\ .$$

For each of these polynomials the extrapolated value $\tilde{T}_{ik}(0)$ is in general a better approximation to the integral I than the interpolated values $T(h_j)$. Since $\tilde{T}_{mm}(h)$ interpolates *all* trapezoidal sums $T(h_j),\ j = 0, \ldots, m$, we can expect $\tilde{T}_{mm}(0)$ to be the most accurate approximation of these.

The extrapolated values $\tilde{T}_{ik}(0)$ can be easily computed by use of the Neville-Aitken algorithm (see [309]) without having to determine the interpolating polynomials themselves. We introduce the abbreviations

$$T_{ik} := \tilde{T}_{ik}(0)$$

for the extrapolated values.

Now, $T_{i0} = T(h_i), i = 0, \ldots, m$ and with the Neville-Aitken algorithm (see [309]) we can compute the other values from

$$T_{ik} = T_{i,k-1} + \frac{T_{i,k-1} - T_{i-1,k-1}}{\left(\dfrac{h_{i-k}}{h_i}\right)^2 - 1}, \quad 1 \le k \le i \le m\ . \tag{10.19}$$

This computation can be arranged in the well-known triangular Romberg-table or T-table (10.20) where the first column contains the values $T_{i,0} = T(h_i)$ of the trapezoidal sums and the following columns are computed by the Neville-Aitken algorithm

(10.19). Thus T_{ik} is contained in row i and column k (if numbering of both is started from zero).

$$\begin{array}{ccccccccc}
T_{0,0} & & & & & & & & \\
& \searrow & & & & & & & \\
T_{1,0} & \to & T_{1,1} & & & & & & \\
& \searrow & & \searrow & & & & & \\
T_{2,0} & \to & T_{2,1} & \to & T_{2,2} & & & & \\
& \searrow & & \searrow & & \searrow & & & \\
\vdots & & \vdots & & \vdots & & \ddots & & \\
& \searrow & & \searrow & & \searrow & & \searrow & \\
T_{m,0} & \to & T_{m,1} & \to & T_{m,2} & \to & \cdots & \to & T_{m,m}
\end{array} \tag{10.20}$$

Each $T_{ik},\ i,k \geq 1$, in row i and column k is computed using the values $T_{i-1,k-1}$ and $T_{i,k-1}$ from the left column $k-1$ and the same and the previous rows i and $i-1$ as indicated by the arrows in (10.20).
In our example program we will use for the step sizes the so-called Romberg sequence, which is obtained by successive halvings of $h_0 = b - a$, i.e.

$$h_0 := b - a, \quad h_i := \frac{h_{i-1}}{2} \ .$$

For this special sequence (10.19) becomes

$$T_{ik} = T_{i,k-1} + (T_{i,k-1} - T_{i-1,k-1})/(4^k - 1) \ . \tag{10.21}$$

Another widely used step size sequence is the Bulirsch sequence

$$h_0 := b - a, \quad h_1 := \frac{h_0}{2}, \quad h_2 := \frac{h_0}{3}, \quad h_i := \frac{h_{i-2}}{2}, \quad i \geq 3$$

which has the advantage that for increasing i the number of subintervals and thus the number of function evaluations does not increase as fast as for the Romberg sequence. Thus the computational cost is lower for the Bulirsch sequence. Nevertheless, we stay with the Romberg sequence since the program will be easier to be read for this case. Also, with the Bulirsch sequence some unnecessary roundoff errors are introduced because of the division by 3 for h_2 which affects all following h_{2j}. This will usually be a minor source for errors only, but it can be reduced by the slightly modified sequence

$$h_0 := b - a, \quad h_1 := \frac{h_0}{2}, \quad h_2 := \frac{5h_0}{16}, \quad h_i := \frac{h_{i-2}}{2}, \quad i \geq 3$$

which can mostly be computed without roundoff errors in floating-point systems with base 2 or 16.
Now we turn to the validated computation of the approximation. Since we can use any of the T_{ik}'s as an approximation for I and since for each of them we will later give an explicit representation of the corresponding remainder term R we will compute enclosures for all T_{ik} by a suitable interval evaluation of the Romberg table (10.20).

One possibility to do this would be to use the representation (10.3) for each of the T_{ik}, i.e. to compute the weights $w_j = w_j^{(ik)}$ corresponding to the approximation T_{ik} and to evaluate (10.3) by use of a long accumulator. This has been done e.g. in [165], [168], however this requires the computation and storage of many weights which, in addition, depend also on the chosen step size sequence.

Here we choose a different approach, which computes the T-table by use of interval arithmetic, but still uses the long accumulator to accumulate the function values. Therefore the arrangement of our computation of the T-table differs somewhat from what is usually proposed in numerical text books. In most text books the computation of $T(h_i)$ is arranged as follows:

$$T(h_i) = \frac{1}{2}T(h_{i-1}) \ + \ h_i(f(a+h_i) + f(a+3h_i) + \cdots + f(b-h_i)) \ .$$

This, however, is not well suited for interval evaluation, since it introduces too many unnecessary roundings: $T(h_{i-1})$ is a sum which has been rounded to an interval already, and adding the second sum multiplied by h_i adds at least three more roundings and many more if this sum is computed in ordinary interval arithmetic only, i.e. without a long accumulator.

Therefore, we proceed by using two long accumulators, i.e. variables *lo* and *up* of type `DOTPRECISION` in the PASCAL-XSC program, which hold the accumulated values of the trapezoidal sums $T(h_i)$ divided by the step size h_i, *lo* for the lower and *up* for the upper bound:

$$\frac{T(h_i)}{h_i} \ \in \ [lo, up]_i := \ \sum_{j=0}^{n}{}^{T} \ 2 \diamond f(x_j)$$

For increasing values of i the additionally computed function values are further accumulated into $[lo, up]$. At the i-th stage the value of $[lo, up]_i$ is rounded once to a floating-point interval and multiplied with the current step size h_i to obtain an enclosure for $T_{i,0}$:

$$T_{i,0} \ \in \ [T_{i,0}] := h_i \diamond [lo, up]_i \ .$$

These enclosing intervals $[T_{i,0}]$ are the values which are actually used for the 0-th column in the T-table. Thus for the starting column we have avoided any unnecessary roundoff errors.

The computation now can be completed by computing an enclosure for each element in the T-table following equation (10.19) or (10.21) for our step size sequence. This computation has to be carried out in interval arithmetic, of course. Thus we obtain enclosures $[T_{i,k}]$, $1 \le k \le i \le m$, for all T-table elements.

The recursive nature of the T-table might give rise to the impression that severe overestimations of the T-table elements may occur. That this will usually not be the case can be seen from the following heuristic case study. Assume, that all enclosures for the trapezoidal sums $T(h_i) = T_{i,0}$ have approximately the same diameter d_0. This assumption is reasonable if most values of f are roughly of the same order of magnitude. This diameter d_0 will be amplified through the recursive computation (10.21) to the diameter d_1 for the elements $T_{i,1}$ in the next column

$$d_1 = d_0 + (d_0 + d_0)/(4-1) = \frac{5}{3} d_0 = \frac{4+1}{4-1} d_0 \; .$$

Continuing the extrapolation we arrive at

$$d_m = c_m d_0 \quad \text{with} \quad c_m = \frac{4+1}{4-1} \cdot \frac{4^2+1}{4^2-1} \cdots \frac{4^m+1}{4^m-1} \; .$$

The amplification factor c_m, however, is rather small here. Some values are:

m	1	2	3	4	5	6	7	10	20
c_m	1.6666	1.8888	1.9488	1.9641	1.9679	1.9689	1.9691	1.9692	1.9692

which shows that the amplification of the diameters is almost negligible. In practice there will also occur some additional roundoff errors, but many examples show that almost never more than one order of magnitude will be lost.
In contrast, however, if we do not compute the trapezoidal sums carefully, then very soon we can loose two or more orders of magnitude up to the last extrapolation.
We close this section with the presentation of the remainder terms R_{ik} which correspond to the approximation terms T_{ik} of the T-table. From [309] we have:

$$\begin{aligned} & I = T_{ik} + R_{ik} \quad \text{with} \\ & R_{ik} = -(b-a) h_{i-k}^2 h_{i-k+1}^2 \cdots h_i^2 \frac{B_{k+1}}{(2k+2)!} f^{(2k+2)}(\xi), \quad \xi \in [a,b] \end{aligned} \tag{10.22}$$

where the h_j are the step sizes and B_k are the Bernoulli numbers. These can be defined by use of the Bernoulli polynomials (see [309]). The first few of the B_k are:

$$\begin{array}{lllllllll} B_1 & = & \frac{1}{6}, & B_2 & = & \frac{1}{30}, & B_3 & = & \frac{1}{42}, \\ B_4 & = & \frac{1}{30}, & B_5 & = & \frac{5}{66}, & B_6 & = & \frac{691}{2730}, \\ B_7 & = & \frac{7}{6}, & B_8 & = & \frac{3617}{510}, & B_9 & = & \frac{43867}{798} \\ B_{10} & = & \frac{174611}{330}, & B_{11} & = & \frac{854513}{138} & B_{12} & = & \frac{236364091}{2730} \; . \end{array} \tag{10.23}$$

We will use only the diagonal entries of the T-table for our algorithm. Thus with the Romberg sequence (10.2.1) as step size sequence the diagonal remainder terms R_{ii} from (10.22) become

$$\begin{aligned} R_{ii} &= -(b-a) h_0^2 \cdots h_i^2 \frac{B_{i+1}}{(2i+2)!} f^{(2i+2)}(\xi) \\ &= -\frac{(b-a)^{(2i+3)}}{2^{i(i+1)}} B_{i+1} \frac{f^{(2i+2)}(\xi)}{(2i+2)!} \; , \quad i = 0, \ldots, m \; . \end{aligned}$$

This is a representation of the R_{ii} which is very similar to the remainder terms for the trapezoidal and Simpson's rule. In order to get enclosures $[R_{ii}]$ we evaluate R_{ii} analogously: ξ is replaced by the interval $[a, b]$ and the derivative of f is computed by automatic differentiation using interval arithmetic. An enclosure $[R_{ii}]$ of R_{ii} thus can be obtained as

$$[R_{ii}] := -(b-a)^{2i+3} \diamond 2^{i(i+1)} \diamond B_{i+1} \diamond (f([a,b]))_{2i+2} \tag{10.24}$$

where $\diamond(f([a,b]))_{2i+2}$ is the Taylor coefficient of order $2i + 2$ of f evaluated in interval arithmetic over $[a, b]$.
Now we are in principle ready to compute enclosures for I by use of Romberg integration: if an absolute error parameter `abs_err` is given, we compute the $[R_{ii}]$ for increasing i until we find an index m where the diameter of $[R_{mm}]$ is less than `abs_err`. Then we may use this m to compute an enclosure $[T_{mm}]$ for T_{mm} and finally we have $I \in [T_{mm}] + [R_{mm}]$.
However, we will get a much more powerful algorithm if we embed this simple algorithm in a framework where also the step size may be adapted. This will be the subject of the next Section.

10.2.2 An Adaptive Strategy

In this Section we will present a simple procedure to adapt the step sizes of the integration to the behaviour of the integrand f. This makes the Romberg integration more powerful and demonstrates how our verifying algorithms may be embedded in a more sophisticated and more efficient framework.
The idea underlying the step size adaption is very simple. Assume we want to obtain an enclosure of I with an absolute error of at most `abs_err`. Then we proceed with a simple bisection method as follows:

1. Compute:
First, fix a number m_{max} to be the maximal count of extrapolations being performed on any subinterval. If the required accuracy can be satisfied by using some $m \leq m_{max}$ then the integral is computed by ordinary Romberg integration with m extrapolations on this subinterval. To find such an m we only have to inspect the diameters of the enclosures of the remainder terms $[R_{ii}]$, $i = 0, ..., m_{max}$.

2. Bisect:
If the error is too large even for $m = m_{max}$ then we bisect the interval and apply our adaptive strategy to both subintervals recursively thereby requiring an absolute error of 0.5*`abs_err` on each subinterval.
To avoid infinite recursion in the bisection step, we force the maximal depth of recursions to stay below some given level `max_depth`. If this level is reached without satisfying the error bounds, then, usually, the whole integral can be computed only with bounds being wider than the required `abs_err`.

Many textbooks recommend to stop the computation of the T-table after approximately seven extrapolations. Following this suggestion, we fix $m_{max} := 7$. For the maximal level for recursions we choose `max_depth` := 20 which allows up to 20 bisections and thus a variation of the step size of six orders of magnitude.
This strategy is certainly not an optimal one and also the choice of the two parameters m_{max} and `max_depth` is only very heuristic. However, the resulting algorithm can be used to compute enclosures for rather complicated and difficult integrals as will be shown in Section 10.2.6.
For more detailed and efficient adaptive verifying methods see [84], [85], [89], [165], [168].

10.2.3 Algorithms

Following the discussion in the previous two Sections 10.2.1 and 10.2.2 we present here an adaptive integration algorithm `INTEGRAL` based on Romberg extrapolation. The input for the algorithm are the integrand f, the interval of integration $[a, b]$, and an absolute error tolerance `abs_err`. The output are an enclosure of the integral (as function result) and an error code indicating if the required absolute error tolerance could be satisfied or not.
In order to keep the algorithm short we break it into three smaller algorithms where only the third one deals with the step size adaption. The first algorithm, `REMAINDER`, computes enclosures for all remainder terms $[R_{ii}]$ (see 10.24) over the interval $[a, b]$ up to some specified maximal index and returns the first index m for which the diameter of $[R_{mm}]$ is less than the prescribed absolute error bound `abs\_err`.
The second algorithm, `ROMBERG`, computes enclosures for the T-table elements up to the index m which was computed by algorithm `REMAINDER`.
Finally, algorithm `INTEGRAL` calls `REMAINDER` to determine if for a given interval `ROMBERG` can be used to compute an enclosure or if the interval has to be bisected. In the latter case `INTEGRAL` calls itself recursively with the two intervals resulting from bisection and with an error requirement of 0.5*`abs\_err`.

Algorithm 10.3: Remainder (f, R, m) {procedure}

{Computation of enclosures for Romberg remainder terms}
input: function f, index m for max. extrapolation count
output: first index m of remainder R_{mm} with diam less than *abs_err*, corresponding enclosure R of R_{mm}.
global variables: integration bounds a, b, error bound *abs_err*, these are local variables to algorithm Integral (10.10.5)

$i := 0$
repeat
  $i := i + 1$
  $h := b - a$
  $R := -h^{2i+3} \cdot 2^{i(i+1)} \cdot B_{i+1} \diamondsuit (f([a, b]))_{2i+2}$

until $(i = m)$ **or** $\text{diam}(R) \leq abs\_err$
return i, R

Algorithm 10.4: Romberg (f, m) {function}

{Computation of enclosures for the Romberg-table}
input: function f, index m for extrapolation count
output: enclosure of T_{mm} (as function value)
global variables: integration bounds a, b, these are local variables to algorithm Integral (10.10.5)

$h := b - a$
$l := 1$
$[lo, up] := 0.5 \cdot (f(a) + f(b))$
$t_0 := h \cdot \Diamond[lo, up]$
for $i := 1$ **to** m **do** {compute trapezoidal sums}
  $h := 0.5 \cdot h$
  $[lo, up] := [lo, up] + \sum_{j=1}^{l} \Diamond f(a + (2j-1)h)$
  $t_i := h \cdot \Diamond[lo, up]$
  $l := 2 \cdot l$
$q := 1$
for $i := 1$ **to** m **do** {compute Romberg table}
  $q := 4 \cdot q$
  for $j := m$ **downto** i **do** $t_j := t_j + (t_j - t_{j-1})/(q-1)$
Romberg $:= t_m$

Algorithm 10.5: Integral $(f, a, b, abs\_err, error)$ {function}

{Romberg Integration with Step Size Adaption and Verification}
input: function f, integration bounds a, b, error bound abs_err
output: error indicator, function value (enclosure of integral value)

(a) $depth := depth + 1$ {increase recursion level}

(b) $m := m\_max$ {maximum extrapolation count}

(c) Remainder (f, R, m) {compute remainder R and extrpolation count m}

(d) **if** $(\text{diam}(R) > abs\_err)$ **and** $(m = m\_max)$ **and** $(depth \leq max\_depth)$
  then {bisection: recursive calls of Integral :}
    $integral\_value :=$ Integral $(f, a, \text{mid}([a, b]), abs\_err/2, error)$
      $+$ Integral $(f, \text{mid}([a, b]), b, abs\_err/2, error)$
  else {use T-table and remainder:}
    $integral\_value :=$ Romberg $(f, m) - R$

(e) Integral $:= integral\_value$ {function result}

(f) {determine accuracy:}
  if $\text{diam}(integral\_value) \leq abs\_err$ **then** $error := 0$
  else $error := 1$

(g) $depth := depth - 1$ {decrease recursion level}

(h) **return** $error$, Integral

10.2.4 Applicability of the Algorithms

Here basically the same restrictions and comments hold which have already been discussed in Section 10.1.3.
However, since we have implemented an adaptive method we can integrate more complicated functions here, than with the simple programs for the trapezoidal and Simpson's rule.

10.2.5 PASCAL-XSC Program Code

```
MODULE iromberg;

USE i_ari,itaylor;

CONST m_max =  7;    { Max. count of extrapolations. If m_max is    }
                     { increased, then the following array Ber must }
                     { be initialized with more data in the module  }
                     { body                                         }

      max_depth = 20;  { max. allowed recursive calls for function }
                       { INTEGRAL, i.e. integration step size may  }
                       { vary by ~ 6 orders of magnitude (~ 2↑20)  }

VAR Ber   : IVECTOR[1..m_max+1];  { Array for Bernoulli numbers,    }
                                  { initialized in the module body }
    depth : INTEGER;


FUNCTION POWER( x : INTERVAL; i : INTEGER ) : INTERVAL;
{-------------------------------------------------------------------}
{ power function :  interval ↑ integer,                             }
{                   faster than runtime function interval↑interval }
{ only valid for i > 0 !!                                           }
{-------------------------------------------------------------------}
VAR r,s : INTERVAL;
BEGIN
  r:= 1.0;
  s:= x;
  WHILE i>0 DO
  BEGIN
     IF odd(i) THEN r:= r*s;
     s:= sqr(s);
     i:= i div 2;
  END;
  power:= r;
END;

{--------------------------------------------------------------------------}
{                                                                          }
{ The function INTEGRAL contained in iromberg computes an enclosure for  }
{ the value of the definite integral of a function f(x) over the interval}
{ [a,b] by use of the Romberg extrapolation method.                       }
```

```
{                                                                    }
{ iromberg uses the operators and functions of the module itaylor for the}
{ computation of the higher derivatives of f in the remainder term by use}
{ of automatic differentiation.                                      }
{                                                                    }
{ The function INTEGRAL uses a simple adaptive strategy to refine the    }
{ interval of integration: if the prescribed absolute error bound abs_err}
{ cannot be satisfied on the interval [a,b], then INTEGRAL bisects [a,b] }
{ and calls itself recursively for each of both subintervals with a      }
{ modified absolute error request of abs_err/2 for each subinterval.     }
{                                                                    }
{ The maximum number of nested recursive calls is limited by the constant}
{ max_depth ( = 20 currently ).                                      }
{ The maximum number of extrapolations on each subinterval is limited by }
{ the constant m_max ( = 7 currently ).                              }
{                                                                    }
{ The parameters of the global function integral are                 }
{   f       : function to be integrated, argument and result as itaylor  }
{   a,b     : integration bounds                                     }
{   abs_err : upper bound for the absolute error required from user      }
{   error   : error indicator upon return of function  integral          }
{             error = 0  result meets required accuracy              }
{             error = 1  result has diameter > abs_err               }
{                                                                    }
{                                                     b              }
{   function's result : interval enclosing  integral  f(x) dx            }
{                                             a                      }
{--------------------------------------------------------------------}

GLOBAL FUNCTION INTEGRAL( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                             a,b     : REAL;
                             abs_err : REAL;
                         VAR error   : INTEGER )  : INTERVAL;

PROCEDURE remainder( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                     VAR R : INTERVAL;
                     VAR m : INTEGER );
{--------------------------------------------------------------------}
{ Computes on the interval [a,b] enclosures for the remainder terms R_ii }
{ of the diagonal of the Romberg-table, i = 1..m, where m = m_max ini-   }
{ tially. The first index i<=m for which the remainder term has diameter }
{ less than abs_err determines the extrapolation count and is returned   }
{ as new m. If no such value exists then m=m_max will be unchanged.      }
{ In any case, the remainder term R_mm corresponding to m will be        }
{ returned in parameter R.                                           }
{--------------------------------------------------------------------}
VAR fh : itaylor[0..2*m+2];
    h  : INTERVAL;
    i  : INTEGER;
BEGIN
   expand( fh, intval(a,b) );
   fh:= f( fh );
   h := intval(b) - intval(a);
   i := 0;
   REPEAT
      i:= i+1;
      R:= power(h,2*i+3)/power(intval(2),i*(i+1)) * Ber[i+1] * fh[2*i+2];
   UNTIL ( i = m ) OR ( diam(R) <= abs_err );
   m:= i;
END;

FUNCTION romberg( FUNCTION f( x : itaylor ) : itaylor[0..ub(x)];
                  m : INTEGER ) : INTERVAL;
{--------------------------------------------------------------------}
```

```
{ Computes the T-table of the Romberg extrapolation up to the diagonal   }
{ element T_mm. The trapezoidal sums T_i0 are stored initially in the    }
{ array t as t[i],i = 0,...,m. The elements of t are then overwritten by }
{ the extrapolated values T_ik. Finally, t[i] holds the values T_ii of   }
{ the diagonal of the T-table, i = 0,...,m.                              }
{ The trapezoidal sums T(h_i) are accumulated in two dotprecision variab-}
{ les lo and up (lower and upper bounds) and then rounded to t[i]. The   }
{ extrapolation is computed in ordinary interval arithmetic.             }
{ The functions result is T_mm = t[m].                                   }
{------------------------------------------------------------------------}
VAR i,j,l : INTEGER;
    h     : INTERVAL;
    q     : REAL;
    fa,fb : itaylor[0..0];
    lo,up : DOTPRECISION;
    t     : IVECTOR[0..m];
BEGIN
   h:= intval(b)-intval(a);
   l:= 1;

   { computation of trapezoidal sums : }
   expand( fa, intval(a) );
   fa:= f( fa );
   expand( fb, intval(b) );
   fb:= f( fb );
   lo:= #( 0.5*inf(fa[0]) + 0.5*inf(fb[0]) );
   up:= #( 0.5*sup(fa[0]) + 0.5*sup(fb[0]) );

   t[0]:= h * intval( #<(lo), #>(up) );

   FOR i:= 1 TO m DO
   BEGIN
      { initialize column of trapezoidal sums : }
      h:= 0.5*h;
      FOR j:= 1 TO l DO BEGIN
                           expand( fa, intval(a)+h*(2*j-1) );
                           fa:= f( fa );
                           lo:= #( lo + inf(fa[0]) );
                           up:= #( up + sup(fa[0]) );
                        END;

      t[i]:= h * intval( #<(lo), #>(up) );
      l   := 2*l;
   END;

   { extrapolation part : }
   q:= 1.0;
   FOR i:= 1 TO m DO
   BEGIN
      { extrapolation : }
      q:= 4.0*q;
      FOR j:= m DOWNTO i DO  t[j]:= t[j] + ( t[j] - t[j-1] ) / (q-1.0);
   END;
   romberg:= t[m];
END;

VAR m                : INTEGER;
    integral_value,R : INTERVAL;
    mid_point        : REAL;

BEGIN  { INTEGRAL }
   depth:= depth + 1;     { increase recursion level }
   m:= m_max;
   remainder( f, R, m );  { compute remainder R and determine }
```

```
                         { extrapolation count m              }

   mid_point:= mid( intval(a,b) ); { if [a,b] has to be bisected }

   IF ( diam(R) > abs_err ) AND ( m = m_max ) AND ( depth <= max_depth )
      THEN
        { bisection by recursive calls, forget remainder R : }
        integral_value:= integral( f, a, mid_point, 0.5*abs_err, error )
                       + integral( f, mid_point, b, 0.5*abs_err, error )
      ELSE
        { compute T-table and use remainder R : }
        integral_value:= romberg( f, m ) - R;

   IF diam(integral_value) <= abs_err THEN error:= 0
                                      ELSE error:= 1;
   INTEGRAL:= integral_value;

   depth:= depth - 1;     { decrease recursion level }
END;   { INTEGRAL }


BEGIN
   { initialize counter for recursion depth of function INTEGRAL : }
   depth:= 0;

   { initialize Bernoulli numbers : }
   Ber[ 1]:=      intval(1.0)/  6.0;   Ber[ 2]:=      intval(1.0)/  30.0;
   Ber[ 3]:=      intval(1.0)/ 42.0;   Ber[ 4]:=      intval(1.0)/  30.0;
   Ber[ 5]:=      intval(5.0)/ 66.0;   Ber[ 6]:=    intval(691.0)/2730.0;
   Ber[ 7]:=      intval(7.0)/  6.0;   Ber[ 8]:=   intval(3617.0)/ 510.0;

{ If m_max > 7 then more Bernoulli numbers will be needed : }
{
   Ber[ 9]:=  intval(43867.0)/798.0;   Ber[10]:= intval(174611.0)/ 330.0;
   Ber[11]:= intval(854513.0)/138.0;   ... etc.
}
END.
```

10.2.6 Test Results

Since we have written our Romberg algorithm in a more sophisticated way by applying an adaptive strategy of subinterval generation we can try to integrate more difficult functions f than with our simple trapezoidal and Simpson routines. The first example is the same as for the trapezoidal and Simpson's rule, i.e. the integral $I(c)$ from (10.18), here with $c = 10$ only. It shows that our adaptive Romberg algorithm is much more efficient than the simple trapezoidal and Simpson rules. The second examples is a slight modification of one of Bulirsch, [77], which can also be found in [165], [168] and the third one is another example from [165].

Example 10.1:

Again we consider the integral $I(10)$ from (10.18). The results which are listed in Table 10.5 were computed with the following short PASCAL-XSC program and demonstrate the superiority of our adaptive Romberg algorithm over the simple trapezoidal and Simpson rules. The columns of Table 10.5 have the same meaning as in Section 10.1.5, n being the total number of function evaluations over all subintervals.

```
PROGRAM example-romberg1;

USE i_ari,itaylor;
USE iromberg;


FUNCTION f ( x : itaylor ) : itaylor[lb(x)..ub(x)];
BEGIN
   f := 100 / ( 1 + sqr(10*x) );
END;

VAR a,b,abs_err : REAL;
    error       : INTEGER;
    int         : INTERVAL;

BEGIN
   write( 'a,b = ' ); read(a,b);
   REPEAT
      write( 'abs_err = ' ); read( abs_err );
      IF abs_err > 0 THEN
      BEGIN
         int:= integral( f, a,b, abs_err, error );
         writeln( int, '  ', diam(int):15, error:3 );
      END;
   UNTIL abs_err <= 0;
END.
```

abs_err	$[I(10)]$	$d([I(10)])$	n	error
1E+00	$[29, 30]$	5.66E-01	160	0
1E-01	2.9_3^5	3.84E-02	288	0
1E-02	2.942_1^4	1.04E-03	448	0
1E-04	2.94225_5^6	6.24E-06	896	0
1E-06	2.942255_3^4	2.93E-07	576	0
1E-08	2.9422553348_5^7	7.85E-10	832	0
1E-10	$2.942255348 60_7^8$	2.78E-12	1536	0
1E-12	$2.9422553486 07_4^5$	3.88E-13	1728	0
1E-14	2.9422553486074_6^8	8.53E-14	2048	1

Table 10.5: Results for I(10) with adaptive Romberg algorithm

Example 10.2:

The integral to be enclosed is

$$I_2 = \int_0^2 2xe^{x^2} \sin(e^{x^2}) dx \ . \tag{10.25}$$

The difficulty with this integrand $f(x)$ is that after a very smooth part between $x = 0$ and $x \approx 1$ the function continues with increasingly wild oscillations with rapidly growing amplitude and frequency. Figure 10.1 shows the graph of this integrand.

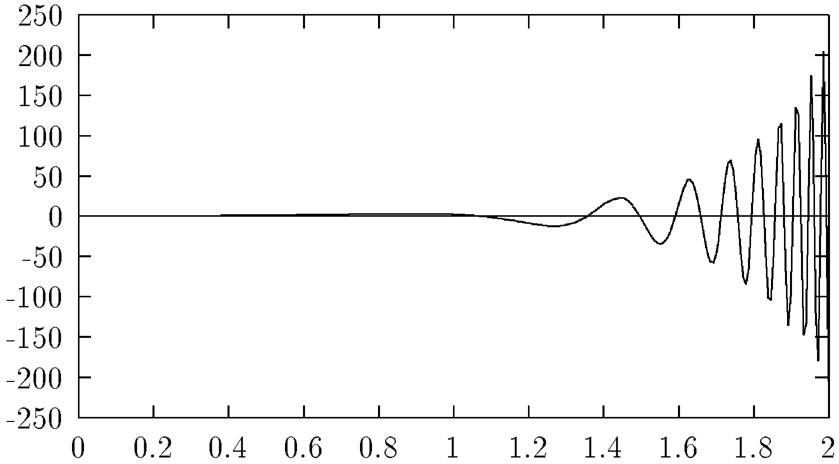


Figure 10.1: $f(x) = 2xe^{x^2} \sin(e^{x^2})$

The integral in (10.25) can be computed exactly as:

$$I_2 = -\cos(e^{x^2})|_0^2 = \cos 1 - \cos e^4 \in 0.910964039265932_8^9$$

We use the following short PASCAL-XSC program to compute enclosures for I_2 with the Romberg module for different choices of the error parameter `abs_err`.

```
PROGRAM example-romberg2;

USE i_ari,itaylor;
USE iromberg;


FUNCTION f ( x : itaylor ) : itaylor[lb(x)..ub(x)];
VAR ex : itaylor[lb(x)..ub(x)];
BEGIN
   ex:= exp(sqr(x));
   f  := 2*x*ex*sin(ex);
END;

VAR a,b,abs_err : REAL;
    error       : INTEGER;
    int         : INTERVAL;

BEGIN
   write( 'a,b = ' ); read(a,b);
   REPEAT
```

```
      write( 'abs_err = ' ); read( abs_err );
      IF abs_err > 0 THEN
      BEGIN
         int:= integral( f, a,b, abs_err, error );
         writeln( int, '  ', diam(int):15, error:3 );
      END;
   UNTIL abs_err <= 0;
END.
```

From the program we get the enclosures $[I_2]$ in the following Table 10.6. Again the columns in Table 10.6 have the same meaning as in the previous examples.

abs_err	$[I_2]$	$d([I_2])$	n	error
1E+00	$[0.83, 0.99]$	1.48E-01	304	0
1E-01	0.9_0^2	8.87E-03	352	0
1E-02	0.9_{08}^{14}	4.38E-03	416	0
1E-04	0.9109_5^8	1.95E-05	640	0
1E-06	0.91096_{39}^{41}	9.94E-08	800	0
1E-08	0.9109640_{38}^{40}	9.98E-10	928	0
1E-10	0.9109640392_5^8	2.64E-11	1376	0
1E-12	0.91096403926_{56}^{62}	5.12E-13	1728	0
1E-14	0.91096403926_{57}^{62}	3.73E-13	2496	1

Table 10.6: Results for I_2 with adaptive Romberg algorithm

Example 10.3:

The integral to be enclosed in this third example is

$$I_3 = \int_0^4 \left(\frac{1}{a^2+(3x-1)^2} - \frac{1}{a^2+(3x-4)^2} + \frac{1}{a^2+(3x-7)^2} - \frac{1}{a^2+(3x-10)^2}\right)dx$$

where $a = 0.1$. This is another difficult integral since the integrand has four sharp peaks with maximal values around +100 and -100 and is much smaller between these peaks. See Figure 10.2. Again, this integral can be computed exactly (as sum of several arctan values). We only give an enclosure of I_3 obtained from the exact representation:

$$I_3 \in -0.151963942232930_5^6$$

The graph of the integrand is shown in the following Figure 10.2,
We use the following short PASCAL-XSC program to compute enclosures for I_3 with the Romberg module for different choices of the error parameter `abs_err`.

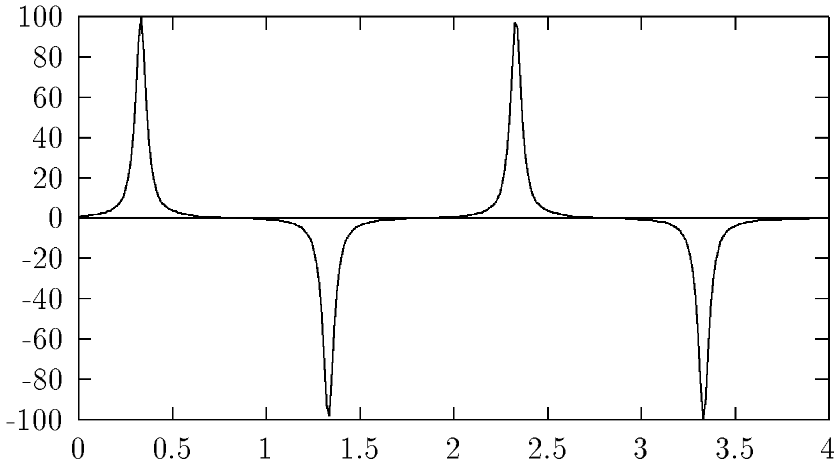


Figure 10.2: $f(x) = \frac{1}{a^2+(3x-1)^2} - \frac{1}{a^2+(3x-4)^2} + \frac{1}{a^2+(3x-7)^2} - \frac{1}{a^2+(3x-10)^2}, \quad a = 0.1$

```
PROGRAM example-romberg3;

USE i_ari,itaylor;
USE iromberg;


FUNCTION f ( x : itaylor ) : itaylor[lb(x)..ub(x)];
BEGIN
  f:= 100 * (  1/(1+sqr(30*x-10)) - 1/(1+sqr(30*x-40 ))
             + 1/(1+sqr(30*x-70)) - 1/(1+sqr(30*x-100)) );
END;

VAR a,b,abs_err : REAL;
    error       : INTEGER;
    int         : INTERVAL;

BEGIN
   write( 'a,b = ' ); read(a,b);
   REPEAT
      write( 'abs_err = ' ); read( abs_err );
      IF abs_err > 0 THEN
      BEGIN
         int:= integral( f, a,b, abs_err, error );
         writeln( int, '  ', diam(int):15, error:3 );
      END;
   UNTIL abs_err <= 0;
END.
```

From the program we get the enclosures $[I_3]$ in the following Table 10.7 where the columns have the same meaning as in the previous examples.

All examples show that even our simple adaptive method can already be successfully used to enclose integrals with rather complicated integrands. The great advantage for the subdivision strategy is the fact that we immediately see the accuracy of the

`abs_err`	$[I_3]$	$d([I_3])$	n	`error`
1E+00	$[-0.29, 0.0066]$	2.89E-01	1918	0
1E-01	-0.1_7^3	3.19E-02	1828	0
1E-02	-0.1_6^5	3.66E-03	1384	0
1E-04	-0.1519_7^5	9.36E-06	2080	0
1E-06	-0.15196_{40}^{39}	3.64E-08	4160	0
1E-08	-0.1519639_5^4	2.53E-09	5120	0
1E-10	-0.151963942_3^2	3.51E-11	5888	0
1E-12	-0.15196394223_{31}^{28}	2.23E-13	8640	0
1E-14	-0.15196394223_{31}^{28}	1.64E-13	10944	1

Table 10.7: Results for I_3 with adaptive Romberg algorithm

partial integrals by computing first an enclosure of the corresponding remainder term. For classical approximation algorithms it is much harder to decide how much the error on some subinterval contributes to the error of the whole integral.

10.3 Notes and References

If a verifying version of an integration formula is to be used in the form (10.3) then the nodes x_i and the weights w_i must be represented exactly on the computer or, if this is not possible, enclosures for these quantities must be used. For many classes of integration formulas the weights are rational numbers and the nodes are rational multiples of the end-points a, b. Usually these values can be represented exactly in floating-point or a floating-point enclosure of maximum accuracy can be computed trivially.

However, there are also classes like e.g. Gaussian quadrature where the nodes and weights are irrational and may be obtained by complicated algorithms only. Then, in addition to the construction of a verifying numerical integration algorithm, we also must device algorithms for computing enclosures of the nodes and weights. This has been done e.g. for Gaussian quadrature in [311].

A similar problem which is of minor difficulty, however, arises if the integration bounds a, b are not floating-point numbers. Then these must be enclosed in intervals and the integration must be performed over these boundary intervals separately with a very simple rule (see e.g. [84], [165], [168]. An alternative would be to transform the interval of integration $[a, b]$ to a floating-point interval, $[0, 1]$, say. Then a, b appear as constants in the integrand. However, the evaluation of the transformed integrand could be more sensitive to over-estimations since the bounds a, b are intervals and may appear in many places of the integrand.

More sophisticated integration algorithms do not use equidistant step sizes, but rather adapt the step sizes according to the behaviour of the integrand f. We have demonstrated this already with our simple adaptive Romberg integration. Other

more efficient adaptive integration algorithms with verification have been developed e.g. by [84], [85], [89], [165], [166], [167], [168].
Verifying algorithms on the basis of Romberg extrapolation have also been developed for two-dimensional integrals and these can even further be generalized to multi-dimensional integrals. For more details see Storck, [310], [312].

Appendix A

The Features of PASCAL–XSC

In this chapter, which is taken from *Numerical Toolbox for Verified Computing I* [124], we give a short overview of the new concepts of the programming language PASCAL–XSC, a universal PASCAL eXtension for Scientific Computation with extensive predefined modules for scientific computation. For a complete language reference and examples, we refer to [173] and [174].
PASCAL–XSC is available for personal computers, workstations, mainframes and supercomputers. Its modern language concepts make PASCAL–XSC a powerful tool for solving scientific problems. The mathematical definition of the arithmetic is an intrinsic part of the language, including optimal arithmetic operations with directed roundings that are directly accessible in the language. Further arithmetic operations for intervals and complex numbers and even for vector/matrix operations provided by precompiled arithmetical modules are defined with maximum accuracy according to the general principle of semimorphism (see Sections **??** and **??**).

PASCAL–XSC contains the following features:

- ISO Standard PASCAL
- Universal operator concept (user-defined operators)
- Functions and operators with arbitrary result type
- Overloading of procedures, functions, and operators
- Overloading of the assignment operator
- Module concept
- Dynamic arrays
- Access to subarrays
- String concept
- Controlled rounding
- Optimal (exact) scalar product
- Predefined type *dotprecision* (a fixed-point format to cover the entire range of floating-point products)
- Additional predefined arithmetic types such as *complex, interval, rvector, rmatrix* etc.
- Highly accurate arithmetic for all predefined types

- Highly accurate mathematical functions
- Exact evaluation of expressions (#-expressions)

These new language features are discussed in the following sections.

A.1 Predefined Data Types, Operators, and Functions

PASCAL–XSC adds the following numerical data types to those available in Standard PASCAL:

	complex	*interval*	*cinterval*
rvector	*cvector*	*ivector*	*civector*
rmatrix	*cmatrix*	*imatrix*	*cimatrix*

Here, the prefix letters *r*, *i*, and *c* are abbreviations for *real*, *interval*, and *complex*. Hence, *cinterval* means *complex interval*, *cimatrix* denotes complex interval matrices, and *rvector* specifies real vectors. The vector and matrix types are defined as dynamic arrays and can be used with arbitrary index ranges.

PASCAL–XSC also supplies the data type *dotprecision* representing a fixed-point format covering the entire range of floating-point products. The type *dotprecision* allows scalar results — especially sums of floating-point products — to be stored exactly. It is used in connection with *accurate expressions* (see Section A.7).

Many operators are predefined for these types in the arithmetic modules (see Section A.9). All of these operators, as well as the operators for type *real*, deliver results of maximum accuracy.

PASCAL–XSC provides 11 new operator symbols beyond those provided by Standard PASCAL. These are the operators $\circ\!<$ and $\circ\!>$, with $\circ \in \{+, -, *, /\}$ for operations with downwardly and upwardly directed rounding, and the operators $**$, $+*$, $><$ needed in interval computations for the intersection, the interval hull, and the test for disjointness, respectively.

Tables A.1, A.2, and A.3 show all predefined operators in connection with the possible combinations of operand types. In Tables A.1 and A.3, the symbol $\triangleleft$ is used as an abbreviation: $\triangleleft \in \{=, <>, <, <=, >, >=\}$. The operators of the first row of Table A.2 are monadic, i.e. there is no left operand. Let $\circ \in \{+, -, *, /\}$, and $\bullet \in \{+, -, *\}$. For vector and matrix types, $*$ denotes the scalar or matrix product.

All usual operations, even those in the higher mathematical spaces, have been realized as operators and can be used in conventional mathematical notation to make programs more easily readable.

Table A.1: Predefined basic operators ($\triangleleft \in \{=, <>, <, <=, >, >=\}$)

right operand / left operand	integer	boolean	char	string	**set**
monadic	$+, -$	**not**			
integer	$+, -, *, /$, **div**, **mod**, $\triangleleft$				**in**
boolean		**or**, **and**, $=, <>, <=, >=$			**in**
char			$+$ $\triangleleft$	$+$ $\triangleleft$, **in**	**in**
string			$+$ $\triangleleft$	$+$ $\triangleleft$, **in**	
set					$+, -, *, =, <>, <=, >=$
enumeration type					**in**

Table A.2: Predefined arithmetical operators ($\circ \in \{+, -, *, /\}$, $\bullet \in \{+, -, *\}$)

right operand / left operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
monadic	$+, -$	$+, -$	$+, -$	$+, -$	$+, -$	$+, -$
integer real complex	$\circ, \circ<, \circ>, +*$	$+, -, *, /, +*$	$*, *<, *>$	$*$	$*, *<, *>$	$*$
interval cinterval	$+, -, *, /, +*$	$+, -, *, /, +*, **$	$*$	$*$	$*$	$*$
rvector cvector	$*, *<, *>, /, /<, />$	$*, /$	$\bullet, \bullet<, \bullet>, +*$	$+, -, *, +*$		
ivector civector	$*, /$	$*, /$	$+, -, *, +*$	$+, -, *, +*, **$		
rmatrix cmatrix	$*, *<, *>, /, /<, />$	$*, /$	$*, *<, *>$	$*$	$\bullet, \bullet<, \bullet>, +*$	$+, -, *, +*$
imatrix cimatrix	$*, /$	$*, /$	$*$	$*$	$+, -, *, +*$	$+, -, *, +*, **$

In Table A.3, the operators $<=$ and $<$ denote the subset relations, whereas $>=$ and $>$ denote the superset relations if the operands are interval data types. As already mentioned, $><$ denotes the test for disjointness for interval types. The operator **in** tests for membership of a point in an interval or for strict enclosure of an interval in the interior of another interval. We also call this the inner inclusion relation (see Section **??** for details).

Table A.3: Predefined relational operators ($\lhd \in \{=, <>, <, <=, >, >=\}$)

left operand \ right operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
integer real complex	$\lhd$	**in** =, <>				
interval cinterval	=, <>	**in**, ><, $\lhd$				
rvector cvector			$\lhd$	**in** =, <>		
ivector civector			=, <>	**in**, ><, $\lhd$		
rmatrix cmatrix					$\lhd$	**in** =, <>
imatrix cimatrix					=, <>	**in**, ><, $\lhd$

Compared with Standard PASCAL, PASCAL–XSC provides an extended set of mathematical functions (see Table A.4). These functions are available for the types *integer*, *real*, *complex*, *interval*, and *cinterval* with generic names and deliver results of maximum accuracy.

Table A.4: Predefined mathematical functions

Function	Generic Name	Function	Generic Name
Absolute Value	abs		
Square	sqr	Square Root	sqrt
Exponential Function	exp	Natural Logarithm (Base e)	ln
Power Function (Base 2)	exp2	Logarithm (Base 2)	log2
Power Function (Base 10)	exp10	Logarithm (Base 10)	log10
Sine	sin	Arc Cosine	arccos
Cosine	cos	Arc Sine	arcsin
Tangent	tan	Arc Tangent	arctan
Cotangent	cot	Arc Cotangent	arccot
Hyperbolic Sine	sinh	Inverse Hyperbolic Sine	arsinh
Hyperbolic Cosine	cosh	Inverse Hyperbolic Cosine	arcosh
Hyperbolic Tangent	tanh	Inverse Hyperbolic Tangent	artanh
Hyperbolic Cotangent	coth	Inverse Hyperbolic Cotangent	arcoth

PASCAL–XSC provides type transfer functions *intval*, *inf*, *sup*, *compl*, *re*, and *im* for conversion between and access to the components of the numerical data types (for scalar, vector and matrix types). Also, some additional functions like *diam*, *mid*, or *transp* for computing diameter and midpoint of an interval or the transpose of a matrix are provided.

A.2 The Universal Operator Concept

PASCAL–XSC makes programming easier by allowing the programmer to define functions and operators with arbitrary result type. The advantages of these concepts are illustrated by the simple example of polynomial addition. If we define the type *polynomial* by

```
const max_degree = 20;
type  polynomial = array [0..max_degree] of real;
```

in Standard PASCAL, then the addition of two polynomials is implemented as a *procedure*

```
procedure add ( a, b : polynomial; var c : polynomial );
{ Computes c = a + b for polynomials }
var
  i : integer;
begin
  for i := 0 to max_degree do c[i] := a[i] + b[i];
end;
```

Several calls of *add* have to be used to compute the expression $z = a + b + c + d$:

```
add(a,b,z);
add(z,c,z);
add(z,d,z);
```

In PASCAL–XSC, we define a *function* with the result type *polynomial*

```
function add ( a, b : polynomial ) : polynomial;
{ Delivers the sum a + b for polynomials }
var
  i : integer;
begin
  for i := 0 to max_degree do add[i] := a[i] + b[i];
end;
```

Now, the expression $z = a + b + c + d$ may be computed as

```
z := add(a,add(b,add(c,d)));
```

Even clearer is the *operator* in PASCAL–XSC

```
operator + ( a, b : polynomial ) result_polynomial : polynomial;
{ Delivers the sum a + b for polynomials }
var
  i : integer;
begin
  for i := 0 to max_degree do result_polynomial[i] := a[i] + b[i];
end;
```

Now, the expression may be written in the common mathematical notation

```
z := a + b + c + d;
```

A programmer may also define a new name as an operator. A priority is assigned in a priority declaration.

A.3 Overloading of Procedures, Functions, and Operators

PASCAL–XSC permits the overloading of function and procedure identifiers. A generic name concept allows the programmer to apply the identifiers *sin*, *cos*, *exp*, *ln*, *arctan*, and *sqrt* not only for *real* numbers but also for intervals, complex numbers, or elements of other mathematical spaces. Overloaded functions and procedures are distinguished by number, order, and type of their parameters. The result type is *not* used for distinction.
As illustrated above, operators also may be overloaded. Even the assignment operator (:=) may be overloaded so that the mathematical notation may be used for assignments:

```
operator := ( var p : polynomial; r : real );
var
  i : integer;
begin
  p[0] := r;
  for i := 1 to max_degree do p[i] := 0;
end;

var
  x : real;
  q : polynomial;
begin
  x := 1.5;
  q := x;   { Polynomial with constant value 1.5 }
end.
```

The overloading concept also applies to the predefined procedures *read* and *write* in a slightly modified way. The first parameter of a newly declared input/output procedure must be a **var**-parameter of file type. The second parameter represents the quantity that is to be input or output. All following parameters are interpreted as format specifications. One could provide an overloaded output facility for polynomials as

```
procedure write ( var t : text; p : polynomial; w : integer );
var
  i          : integer;
  PolyIsZero : boolean;   { Signals 'p' is a zero polynomial }
begin
  PolyIsZero := true;
  for i := 0 to max_degree do
    if (p[i] <> 0) then
    begin
      if PolyIsZero then write(t,'  ')  else  write(t,'+ ');
      writeln(t,p[i]:w,' * x↑',i:1);
      PolyIsZero := false;
    end;
  if PolyIsZero then writeln(t,'   0 (= zero polynomial)');
end;
```

The file parameter is omitted from the calling of an overloaded input/output procedure if the standard file *input* or *output* is assumed. The format parameters must be introduced and separated by colons. Moreover, several input or output statements can be combined to a single statement as in Standard PASCAL.

```
var
  r : real;
  p : polynomial;
begin
  r := 2;
  p := 5;
  write(p : 7, r : 10, r/5);
end.
```

A.4 Module Concept

The module concept allows the programmer to separate large programs into modules and to develop and compile them independently of each other. The control of syntax and semantics may be carried out beyond the bounds of the modules. Modules are introduced by the reserved word **module** followed by a name and a semicolon. The body of a module is built up quite similarly to that of a PASCAL program. The significant exception is that the objects to be exported from the module are identified by the reserved word **global** directly in front of the reserved words **const, type, var, procedure, function**, and **operator** and directly after **use**. Moreover, if **global** is placed after the equality sign in a type declaration, then the module exports both the type identifier and the internal structure (e.g. names of components, component types) of the type, which is then called a *non-private* type. Without this second **global**, types are called *private*.
Modules are *imported* into other modules or programs via a **use**-clause. The semantics of the **use**-clause are that all objects declared **global** in the imported module are also known in the importing module or program.
The example of a polynomial arithmetic module illustrates the structure of a module:

```
module poly;

use { Other modules ... }
  utility;

{ Local declarations }
{--------------------}
const
  max_degree = 20;

var
  LocalVariable : integer;

function LocalFunction : real;
var
  Value : real;
begin
  { Do some computations ... }
  LocalFunction := Value;
end;

procedure LocalProcedure;
begin  { Do something ... }  end;
```

```
{ Other local declarations ... }

{ Global declarations }
{---------------------}
global type polynomial = array [0..max_degree] of real;

global procedure read ( var t : text; var p : polynomial );
begin  { Input statements ... }  end;

global procedure write ( var t : text; p : polynomial );
begin  { Output statements ... }  end;

global operator + ( a, b : polynomial ) result_polynomial : polynomial;
{ Delivers the sum a + b for polynomials }
var
  i : integer;
begin
  for i := 0 to max_degree do result_polynomial[i] := a[i] + b[i];
end;

{ Other global declarations ... }

{ Initialization part of the module }
{-----------------------------------}
begin
  LocalVariable := 10;
end. { module poly }
```

A.5 Dynamic Arrays and Subarrays

The concept of dynamic arrays enables the programmer to implement algorithms independently of the length of the arrays used. The index ranges of dynamic arrays are not defined until run-time. Procedures, functions, and operators may be programmed in a fully dynamic manner, since allocation and release of local dynamic variables are executed automatically. Hence, the memory is used optimally.
For example, a dynamic type *dyn_poly* may be declared:

```
type dyn_poly = dynamic array [*] of real;
```

When declaring variables of this dynamic type, the index bounds have to be specified:

```
var p, q : dyn_poly[0..2*n];
```

where the values of the expressions for the index range are computed during program execution. The two functions *lbound(...)* and *ubound(...)* and their abbreviations *lb(...)* and *ub(...)* access the bounds of dynamic arrays which are specified only during execution of the program. The multiplication of two polynomials may be realized dynamically as follows:

```
operator * ( a, b : dyn_poly ) product : dyn_poly[0..ub(a)+ub(b)];
{ Delivers the product a * b of two dynamic polynomials a, b }
var
  i, j   : integer;
  result : dyn_poly[0..ub(a)+ub(b)];
begin
```

```
  for i := 0 to ub(a)+ub(b) do
    result[i] := 0;
  for i := 0 to ub(a) do
    for j := 0 to ub(b) do
      result[i+j] := result[i+j] + a[i] * b[j];
  product := result;
end;
```

A PASCAL–XSC program using dynamic arrays for polynomials follows the template

```
program dynatest (input, output);
type dyn_poly = dynamic array [*] of real;

operator * ( a, b : dyn_poly ) product : dyn_poly[0..ub(a)+ub(b)];
var
  i, j   : integer;
  result : dyn_poly[0..ub(a)+ub(b)];
begin
  { Statements for the computation of the product ... }
  product := result;
end;

procedure read ( var f : text; p : dyn_poly );
begin  { Input statements ... }  end;

procedure write ( var f : text; p : dyn_poly );
begin  { Output statements ... }  end;

procedure dyn_work (degree: integer);
var
  p, q : dyn_poly[0..degree];
  r    : dyn_poly[0..2*degree];
begin
  writeln('Enter p: '); read(p);
  writeln('Enter q: '); read(q);
  r := p * q;
  writeln('p*q = ', r);
end;

var
  actual_degree : integer;

begin  { Main program }
  write('Enter actual degree: ');  read(actual_degree);
  dyn_work(actual_degree);
end.
```

The following example demonstrates that it is possible to access a row or a column of dynamic arrays as a single object. This is called *slice* notation.

```
program slice;
var
  v    : rvector[1..5];
  A    : rmatrix[1..5,1..5];
begin
  v      := A[2];   { 2nd row of A    }
  A[*,3] := v;      { 3rd column of A }
end.
```

A.6 Data Conversion

This section is *critical* to the appropriate use of every routine in this book. The concept is simple: People work in a decimal notation, while computers use a binary representation for numbers. The most common error made by even the most experienced interval guru is to forget that numbers such as 0.1 in input data or as literals in the code are *not* what they seem!
Numerical computations nearly always are executed on non-decimal floating-point systems. The floating-point format usually used is a binary format, so it is inevitable that literal *real* constants must be converted into that data format. This conversion can cause an error. For example, the literal constant 1.1 is not exactly representable in a binary format (see Section **??** for details). To execute this conversion in a controlled way, an additional notation for real literal constants is necessary. While the usual PASCAL notation of *real* numbers implies the conversion with rounding to the nearest floating-point (machine) number, it is possible to specify *real* constants that are converted with rounding to the next-smaller or the next-larger floating-point number by the notations

$$(< \pm\ \mathit{Mantissa}\ \mathrm{E}\ \mathit{Exponent}\ ) \quad \text{and} \quad (> \pm\ \mathit{Mantissa}\ \mathrm{E}\ \mathit{Exponent}\ )\ ,$$

respectively. The E and *Exponent* may be omitted as usual, in which case *Mantissa* must contain a decimal point. The parentheses are mandatory. For example, we can program

```
program round_input;
use i_ari;
var
  x, y : real;
  z    : interval;
begin
  x := (< 1.1);                      { Round 1.1 downwardly       }
  y := (> 1.0E-1);                   { Round 0.1 upwardly         }
  z := intval( (< 0.1), (> 0.1) ); { Enclose 0.1 in an interval }
end.
```

To realize a controlled rounding when entering *real* data from the console or from a text file, the procedures *read* (or *readln*) provide an additional format control parameter r. This *integer* parameter specifies the rounding mode of the *real* value during the input process. The value of a variable x of type *real* can be entered by

```
read(x : r);
```

which causes the value to be rounded (cf. Section **??**):

$r < 0$	round to the next-smaller representable number,
$r = 0$ (or absent)	round to the nearest representable number, or
$r > 0$	round to the next-larger representable number.

A rounding parameter can also be used to convert a string representing a literal *real* constant into a *real* value. Moreover, similar rules apply to the output of *real* values and to their conversion into strings.

A.7 Accurate Expressions (#-Expressions)

The implementation of algorithms with automatic result verification or validation in this book makes extensive use of the accurate evaluation of dot product expressions, i.e. expressions which can be reduced to a single dot product. This includes matrix and vector expressions, where the computation of each component can be reduced to such a dot product form.

To evaluate this kind of expression, the new data type *dotprecision* was introduced. This data type accommodates the full floating-point range with double exponents (see [211] or [217]). *Accurate expressions* (#-expressions) can be formed based on this data type by an accurate symbol (#, #*, #<, #>, or ##) followed by an *exact expression* enclosed in parentheses. The exact expression must have the form of a dot product expression in scalar, vector, or matrix structure and is evaluated without any rounding error. Because of this, the result of an accurate expression is of maximum accuracy in the sense that in every component of the result there is no floating-point number between the exact value and the computed one. That is, the rounded and the exact result differ at most by 1 unit in the last place of the mantissa.

To obtain the unrounded or correctly rounded result of a dot product expression, you need to parenthesize the expression and precede it by the symbol #, optionally followed by a symbol for the rounding mode. Table A.5 shows the possible rounding modes for the dot product expression form.

Table A.5: Rounding modes for accurate expressions

Symbol	Expression Form	Rounding Mode
#*	scalar, vector or matrix	nearest
#<	scalar, vector or matrix	downwards
#>	scalar, vector or matrix	upwards
##	scalar, vector or matrix	smallest enclosing interval
#	scalar only	exact, no rounding

In practice, dot product expressions may contain a large number of terms making an explicit notation very cumbersome. To alleviate this difficulty in mathematics, the symbol $\sum$ is used. For instance, if A and B are n-dimensional matrices, then the evaluation of

$$d = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

represents a dot product expression. PASCAL–XSC provides the equivalent short-hand notation **sum** for this purpose. The corresponding PASCAL–XSC statement for this expression is

```
d := #( for k:=1 to n sum( A[i,k] * B[k,j] ) );
```

where d is a *dotprecision* variable.

Accurate or dot product expressions are used mainly in computing a residual. In the case of a linear system $Ax = b$, $A \in I\!R^{n \times n}$, $x, b \in I\!R^n$, with $Ay \approx b$, an enclosure of the residual $b - Ay$ can be computed as

```
##( b - A * y );
```

We emphasize that there is only one interval rounding operation per component. To get verified enclosures for linear systems of equations, it may be necessary to evaluate an enclosure of the residual $I - RA$ where $R \approx A^{-1}$ and I is the identity matrix. This can be programmed as

```
##( id(A) - R * A );
```

where an interval matrix is computed with only one rounding operation per component. The function *id(...)* generates an identity matrix of appropriate dimension according to the shape of A (see Section A.9).

A.8 The String Concept

The tools provided for handling strings in Standard PASCAL do not allow convenient text processing. For this reason, PASCAL–XSC includes a string concept for the convenient handling of textual information and symbolic computation. With this new data type *string*, you can work with strings of up to *maxint* characters, by specifying a maximum string length less than *maxint* in the declaration part. Thus, a string s declared by

```
var s : string[40];
```

can be up to 40 characters long. The following string operations are available:

- concatenation of strings (operator +)
- actual length of a string (function *length*)
- conversions *string* → *real*, *string* → *integer*, *real* → *string*, and *integer* → *string* (functions *rval*, *ival*, and *image*)
- extraction of substrings (function *substring*)
- position of first appearance of a string in another string (function *pos*)
- comparisons (relational operators <=, <, >=, >, <>, =, and **in**)

A.9 Predefined Arithmetic Modules

The following predefined arithmetic modules are available:

- interval arithmetic (*i_ari*)
- complex arithmetic (*c_ari*)

- complex interval arithmetic (*ci_ari*)
- real matrix/vector arithmetic (*mv_ari*)
- interval matrix/vector arithmetic (*mvi_ari*)
- complex matrix/vector arithmetic (*mvc_ari*)
- complex interval matrix/vector arithmetic (*mvci_ari*)

These modules may be imported via the **use**-statement described in Section A.4. As an example, Table A.6 shows the operators provided by the module for interval matrix/vector arithmetic.

Table A.6: Predefined arithmetic and relational operators from module *mvi_ari*

left operand \ right operand	integer real	interval	rvector	ivector	rmatrix	imatrix
monadic				+, −		+, −
integer real				*		*
interval			*	*	*	*
rvector		*, /	+*	+*, +, −, *, **in**, =, <>		
ivector	*, /	*, /	+*, +, −, *, =, <>	+*, **, +, −, *, **in**, =, <>, ><, <=, <, >=, >		
rmatrix		*, /		*	+*	+*, +, −, *, **in**, =, <>
imatrix	*, /	*, /	*	*	+*, +, −, *, =, <>	+*, **, +, −, *, **in**, =, <>, ><, <=, <, >=, >

In addition to these operators, the module *mvi_ari* provides the following generically named standard operators, functions, and procedures

intval, inf, sup, diam, mid, blow, transp, null, id, read, and *write*.

The function *intval* is used to generate interval vectors and matrices, whereas *inf* and *sup* are selection functions for the infimum and supremum of an interval object. The diameter and the midpoint of interval vectors and matrices are determined via *diam* and *mid*. *Blow* yields an interval inflation, and *transp* is used to get the transpose of a matrix (refer to Chapter **??** for details on the mathematical meaning of these terms).

Zero vectors and matrices are generated by the function *null*, while *id* returns an identity matrix of appropriate shape. Finally, there are the generic input/output procedures *read* and *write*, which may be used in connection with all matrix/vector data types defined in the modules mentioned above.

A.10 Why PASCAL–XSC?

As already said in the introduction of this book, PASCAL–XSC is the language in which all our "tools" are implemented. The reason for this is that its wide range of modern language concepts for scientific computation makes PASCAL–XSC a powerful tool for solving scientific problems and especially well suited as a specification language for programming with automatic result verification. Moreover, PASCAL–XSC is available for personal computers, workstations, mainframes and supercomputers, and so it supports a high degree of portability among very different computers for all our routines, modules, and programs.
see Toolbox I

Appendix B

Additional Remarks Concerning the Data Type *real*

In Section 9.2 the most important features of the data type `real` have been already discussed. We now consider special `real` values as signaling and quiet NaNs in more detail.
By default the PASCAL–XSC runtime assumes that a signaling NaN is identified by bit 51 of the representation of the floating-point number being set. A quiet NaN is identified by bit 51 of the representation of the floating-point number being not set. The sign of a NaN is ignored. The described interpretation of bit 51 is the default setting and may be altered when hardware operations are used for the current installation. Refer to the *local configuration guide* for the actual setting.
A quiet NaN is generated instead of a *real value* if an exception occurred in an operation that does not produce any reasonable arithmetic result due to the exception (invalid operation) and the trap handler is disabled for this exception. The structure of a generated quiet NaN is given in Figure B.1.

bit	63	= 0 or 1	(sign is ignored)
bit	62-52	= 2047	(all bits are set)
bit	51	= 0	(identifies quiet NaN)
bit	32-50	= 0	(reserved)
bit	0-31	= *integer*	(exception code)

Figure B.1: Structure of a quiet NaN

Module *x_real* contains additional constants, types, functions, and procedures for an extended or alternative processing of *real* values (see Section ??). Especially it supports a new data type called `x_ccode` to distinguish different classes of `real` (IEEE exceptions).

B.1 Classification of *real* values (Module *x_real*)

Module *x_real* contains additional constants, types, functions, and procedures for an extended or alternative processing of *real* values (see Section ??). Especially it

supports a new data type called `x_ccode` to distinguish different classes of `real` (IEEE exceptions).
The specification of the *real* data type in section 9.2 suggests a classification of *real* values according to the represented value. A total of 10 different classes of *real* values can be distinguished. The data type *x_ccode* is introduced which enumerates the classification codes using enumeration constants.

```
type x_ccode = ( x_sNaN,  { signaling NaN          }
                 x_qNaN,  { quiet NaN              }
                 x_minf,  { minus infinity         }
                 x_mnor,  { negative normalized    }
                 x_mden,  { negative denormalized }
                 x_mnul,  { minus zero             }
                 x_pnul,  { plus zero              }
                 x_pden,  { positive denormalized }
                 x_pnor,  { positive normalized    }
                 x_pinf   { plus infinity          }
               );
```

```
function x_class ( r : real ) : x_ccode;
```

The return code of function *x_class* is the classification code of type *x_ccode* of the given *real* value argument.

```
function x_value ( c : x_ccode ) : real;
```

The *real* value returned by function `x_value` is a special value corresponding to the given classification code of type *x_ccode*. The returned value for the classification codes are listed in the following table.

code	hexadecimal value	description
`x_sNaN`	`7ff80000fffffff`	signaling NaN
`x_qNaN`	`7ff00000fffffff`	quiet NaN
`x_minf`	`fff0000000000000`	minus infinity
`x_mnor`	`ffefffffffffffff`	negative normalized
`x_mden`	`8000000000000001`	negative denormalized
`x_mnul`	`8000000000000000`	minus zero
`x_pnul`	`0000000000000000`	plus zero
`x_pden`	`0000000000000001`	positive denormalized
`x_pnor`	`7fefffffffffffff`	positive normalized
`x_pinf`	`7ff0000000000000`	plus infinity

Note, that the values returned for `x_sNaN` and `x_qNaN` depend on the installation of your system. Refer to the *local configuration guide*.

B.2 IEEE Exception Handling Routines

In order to manipulate the exception handling environment of IEEE exceptions, a number of constants are defined which can be used together with the PASCAL–XSC procedures `IEEE_environment` and `IEEE_trap_enable`.

```
IEEE_INV_OP
IEEE_DIV_BY_ZERO
IEEE_OVERFLOW
IEEE_UNDERFLOW
IEEE_INEXACT
IEEE_ALL

IEEE_CONTINUE
```

The constants `IEEE_INV_OP`, `IEEE_DIV_BY_ZERO`, `IEEE_OVERFLOW`, `IEEE_UNDERFLOW`, and `IEEE_INEXACT` characterize the five exceptions specified by the IEEE standard. The constant `IEEE_CONTINUE` is used for changing the exception environment.

```
procedure IEEE_environment(action  : integer;
                           handler : integer;
                           mode    : boolean);
```

Procedure `IEEE_environment` transfers an '`action`' code to the embedding environment of an IEEE exception handler which is selected by the *integer* argument 'handler'. The '`mode`' value activates a characterization if it is *true* or inactivates a characterization if it is *false*.
The '`action`' code

```
IEEE_CONTINUE
```

forces the environment to continue processing after the trap handler has terminated. Other codes may be provided by further releases of the PASCAL–XSC runtime system. The trap handler is identified by the *integer* argument '`handler`' which may have one of the values `IEEE_DIV_BY_ZERO`, `IEEE_INEXACT`, `IEEE_INV_OP`, `IEEE_OVERFLOW`, and `IEEE_UNDERFLOW`. The value of `mode` must be *true* in order to activate this characteristic of the exception handling environment. If `IEEE_CONTINUE` is selected and '`mode`' is *false*, then processing is aborted after leaving the trap handler. Otherwise the processing of the program is continued.

```
procedure IEEE_trap_enable(handler : integer; mode : boolean);
```

Procedure `IEEE_trap_enable` sets the enabled status of an IEEE exception handling routine. If the value of `mode` is *true* then the trap handler is enabled. If the value of `mode` is *false*, then the trap handler is disabled. The selection of the trap handler is done by the value of the *integer* argument `handler` which may have one of

the values IEEE_DIV_BY_ZERO, IEEE_INEXACT, IEEE_INV_OP, IEEE_OVERFLOW, and IEEE_UNDERFLOW.
An example for the usage of the procedures IEEE_environment and IEEE_trap_enable is given in Appendix ?? *IEEE Exception Handling Environment.*

```
function IEEE_test(handler : integer) : boolean;
```

Function IEEE_test returns the value of the exception flag that is associated with the IEEE exception identified by 'handler'. The exception flag is not changed by this function. Exceptions flags stay *set* until they are explicitly *reset* via calls to procedure IEEE_reset.
Before processing the first PASCAL–XSC statement all exception flags are reset. Note that the first PASCAL–XSC statement may be placed in an included module and may affect the exception flags before processing the first statement of the program body. Thus, if IEEE exception handling routines are used, it is recommended that all exception flags are explicitly reset in the program body before processing is started.
The exception flag is set explicitly by the user via procedure 'IEEE_set' or automatically by all IEEE floating-point operations if the corresponding exceptional conditions are met. Exception flags are set independent of the enabled status of the related trap handler. Valid numbers for the *integer* argument 'handler' are IEEE_DIV_BY_ZERO, IEEE_INEXACT, IEEE_INV_OP, IEEE_OVERFLOW, and IEEE_UNDERFLOW.

```
procedure IEEE_set(handler : integer);
```

Procedure IEEE_set sets the value of the exception flag that is associated with the IEEE exception identified by 'handler'. Valid numbers for the *integer* argument 'handler' are IEEE_ALL, IEEE_DIV_BY_ZERO, IEEE_INEXACT, IEEE_INV_OP, IEEE_-OVERFLOW, and IEEE_UNDERFLOW. If IEEE_ALL is selected, then all IEEE flags are set.

```
procedure IEEE_reset(handler : integer);
```

Procedure IEEE_reset resets the value of the exception flag that is associated with the IEEE exception identified by 'handler'. Valid numbers for the *integer* argument 'handler' are IEEE_ALL, IEEE_DIV_BY_ZERO, IEEE_INEXACT, IEEE_INV_OP, IEEE_-OVERFLOW, and IEEE_UNDERFLOW. If IEEE_ALL is selected, then all IEEE flags are reset.

```
procedure IEEE_save(var x : integer);
```

Procedure IEEE_save saves the setting of all IEEE exception flags and all IEEE trap enabled flags as value of the integer variable 'x'. The complete flag settings saved in variable 'x' may be restored by calling IEEE_restore with argument 'x'

```
procedure IEEE_restore(x : integer);
```

Procedure IEEE_restore restores the setting of all IEEE exception flags and all IEEE trap enabled flags from the integer value 'x'.

Appendix C

Reliable Error Bounds for Horner's Scheme

We will derive an algorithm to find an a priori error bound for the maximum relative error caused by inexact (not representable) polynomial coefficients and by the usage of machine operations instead of exact operations for the computation of a polynomial using Horner's scheme. It is assumed that a reliable relative error bound for the accuracy of the coefficients of the polynomial is known (for example: all coefficients are computed with maximum accuracy, i.e. they are known to 0.5 units in the last place). Additionally, a relative error bound for the argument x is assumed to be known (in general, x will be the reduced argument and so its machine counterpart $\tilde{x}$ will be afflicted by rounding errors). The usage of an interval arithmetic enables us to compute the desired bound automatically by a computer.

C.1 Theory and Algorithmic Description

Typically we are interested in error bounds for expressions like

$$\begin{aligned}
&\left|\frac{xP_n(x) - \tilde{x} \odot \tilde{P}_n(\tilde{x})}{xP_\infty(x)}\right| \\
&\qquad \leq \ldots \leq \left|P_n(x) - \tilde{P}_n(\tilde{x})\right| \frac{1}{\min\limits_x |P_\infty(x)|} + 1.1 \cdot \{\varepsilon(l) + \varepsilon(x)\}.
\end{aligned} \tag{C.1}$$

Here $P_\infty(x) = \sum_{k=0}^{\infty} a_k x^k$ denotes a series expansion, $P_n(x)$ its n-th partial sum, $\varepsilon(x)$ the relative error of $\tilde{x}$ of x and $\varepsilon(l) := \varepsilon^* = 0.5B^{1-l}$ means Wilkinson's epsilon for an l-digit arithmetic with respect to base B. It is assumed that $\varepsilon(x)$ and $\varepsilon(l)$ are smaller than 10^{-3} . This assumption is almost always fulfilled in practice.

The representation in general $x \cdot P_n(x)$ allows to split off the root of the approximation within the reduced range, (i. e. $\min |P_\infty(x)| > 0$). Thus, a bound for the relative error may be estimated. Most work to be done for the error estimation (C.1) belongs to the computation of a bound for the absolute error $\left|P_n(x) - \tilde{P}_n(\tilde{x})\right|$. This expression will be discussed in the sequel in more detail.

C.1.1 Error Estimation for $\left|P_n(x) - \tilde{P}_n(\tilde{x})\right|$

To estimate the error $\left|P_n(x) - \tilde{P}_n(\tilde{x})\right|$ we split the formula in two parts. The firts part corresponds to inaccuracies inherent in the argument, the second part is introduced by round-off errors when performing the floating-point Horner scheme and errors due to the representation of the polynomial coefficients a_k as finite computer numbers $\tilde{a}_k$.
We are interested in non-negative quantities $\alpha, \beta \in I\!R^+$ with

$$\left|P_n(x) - \tilde{P}_n(\tilde{x})\right| < |P_n(x) - P_n(\tilde{x})| + \left|P_n(\tilde{x}) - \tilde{P}_n(\tilde{x})\right| < \alpha \cdot \varepsilon(x) + \beta \cdot \varepsilon(l). \quad \text{(C.2)}$$

Numerical values of α and β may be computed applying the splitting mentioned above to each individual Horner step. The k-th step of Horner's method (exact, using floating-point arithmetic, using interval arithmetic) is given by:

$$t_n(x) := a_n, \quad \tilde{t}_n(x) := \tilde{a}_n, \quad T_n(X) = [\underline{a}_n, \overline{a}_n] = A_n, \quad k = 0,$$

$$\left.\begin{array}{l} t_{n-k}(x) := t_{n-(k-1)}(x) \cdot x + a_{n-k}, \\ \tilde{t}_{n-k}(x) := \tilde{t}_{n-(k-1)}(x) \odot x \oplus \tilde{a}_{n-k}, \\ T_{n-k}(X) := T_{n-(k-1)}(X) \,\diamondsuit\!\!\!\cdot\, X \,\diamondsuit\!\!\!+\, A_{n-k} \end{array}\right\} \quad k = 1, 2, \ldots, n. \quad \text{(C.3)}$$

After n Horner steps it holds $P_n(x) = t_0(x)$ and $\tilde{P}_n(\tilde{x}) = \tilde{t}_0(\tilde{x})$. $\oplus, \odot$ denote l-digit machine operations, $\diamondsuit\!\!\!\cdot$, $\diamondsuit\!\!\!+$ machine interval operations with l or less than l digits. For the intervals $A_k = [\underline{a}_k, \overline{a}_k]$ and $X = [\underline{x}, \overline{x}]$ it is assumed that $a_k, \tilde{a}_k \in A_k$ and $x, \tilde{x} \in X$, respectively. Moreover, the relative error of $\tilde{a}_k$ is assumed to be less than or equal to $\varepsilon(l)$ (e. g. a_k are computed with maximum accuracy using a multi-precision arithmetic). By construction it holds $t_{n-k}(x)$, $\tilde{t}_{n-k}(x)$, $t_{nik}(\tilde{x})$, $\tilde{t}_{nik}(\tilde{x}) \in T_{n-k}(X)$ for all k and all x, $\tilde{x} \in X$.
In the sequel we need some more notations. The quantities ε_x and ε_l denote some points in the intervals $[-1, 1] \cdot \varepsilon(x)$ and $[-1, 1] \cdot \varepsilon(l)$, respectively. The mapping $\wedge$ is given by

$$\wedge : \left\{ \begin{array}{l} I\!R \to I\!R^+ \\ X \mapsto \hat{X} = \max\{|x| : x \in X\} \end{array} \right. ,$$

and the function ulp is defined by

$$\text{ulp} : \left\{ \begin{array}{l} I\!R \to \{0, 1\} \\ A = [\underline{a}, \overline{a}] \mapsto \text{ulp}(A) = \left\{ \begin{array}{ll} 0, & \underline{a} = \overline{a} \\ 1, & \text{otherwise.} \end{array} \right. \end{array} \right.$$

Using these notations and definitions we now derive algorithms for the computation of the quantities α_{n-k}, $\beta_{n-k} > 0, \quad k = 0, 1, \ldots, n$ with

$$|t_{n-k}(x) - t_{n-k}(\tilde{x})| \le \alpha_{n-k} \cdot \varepsilon(x)$$

and

$$|t_{n-k}(\tilde{x}) - \tilde{t}_{n-k}(\tilde{x})| \le \beta_{n-k} \cdot \varepsilon(l)\,.$$

C.1.2 Computation of the quantities α_{n-k}

We now discuss the computation of the quantities α_{n-k} with $|t_{n-k}(x) - t_{n-k}(\tilde{x})| < \alpha_{n-k} \cdot \varepsilon(x)$. These quantities characterize the error probagation within the individual Horner steps. For $k = 0$ we find immediately

$$|t_n(x) - t_n(\tilde{x})| = |a_n - a_n| = 0, \quad \text{i.e. } \alpha_n := 0. \tag{C.4}$$

and for $k \geq 1$ we get the representation

$$t_{n-k}(\tilde{x}) = t_{n-(k-1)}(\tilde{x}) \cdot \tilde{x} + a_{n-k} = t_{n-(k-1)}(\tilde{x}) \cdot x + t_{n-(k-1)}(\tilde{x}) \cdot x \cdot \varepsilon_x + a_{n-k}\,.$$

Now the estimation

$$\begin{aligned} |t_{n-k}(x) - t_{n-k}(\tilde{x})| &\leq |t_{n-(k-1)}(x) - t_{n-(k-1)}(\tilde{x})| \cdot |x| + |t_{n-(k-1)}(\tilde{x}) \cdot x| \cdot \varepsilon(x) \\ &\leq \alpha_{n-(k-1)} \cdot \hat{X} \cdot \varepsilon(x) + \hat{T}_{n-(k-1)} \cdot \hat{X} \cdot \varepsilon(x), \end{aligned}$$

shows that α_{n-k} can be computed by

$$\alpha_{n-k} := \left(\alpha_{n-(k-1)} + \hat{T}_{n-(k-1)}\right) \cdot \hat{X}. \tag{C.5}$$

That means, starting with $\alpha_n := 0$ we find the error bound α_{n-k} of the k-th step using the bound $\alpha_{n-(k-1)}$ already known of the preceding $(k-1)$-th step according (C.5).

C.1.3 Computation of the quantities β_{n-k}

The quantities β_{n-k} are related to inaccuracies introduced by finite floating-point representations. The computation of these quantities β_{n-k} with $|t_{n-k}(x) - \tilde{t}_{n-k}(x)| \leq \beta_{n-k} \cdot \varepsilon(l)$ can be done in the following way:
For $k = 0$ it holds $|t_{n-k}(x) - \tilde{t}_n(x)| = |a_n - \tilde{a}_n| \leq \hat{A}_n \mathrm{ulp}(A_n) \cdot \varepsilon(l)$, i.e.

$$\beta_{n-k} = \beta_n := \mathrm{ulp} \cdot (A_n) \cdot \hat{A}_n \tag{C.6}$$

and for $k > 0$ we find $\tilde{t}_{n-k}(x) = \tilde{t}_{n-(k-1)}(x) \cdot x + 2.1 \cdot \tilde{t}_{n-(k-1)}(x) \cdot x \cdot \varepsilon_l + \tilde{a}_{n-k} \cdot \varepsilon_l + \tilde{a}_{n-k}$ yielding

$$\begin{aligned} \left|t_{n-k}(x) - \tilde{t}_{n-k}(x)\right| &\leq \left|t_{n-(k-1)} - \tilde{t}_{n-(k-1)}\right| \hat{X} + 2.1 \cdot \hat{T}_{n-(k-1)} \cdot \hat{X} \cdot \varepsilon(l) \\ &\quad + \hat{A}_{n-k} \cdot \varepsilon(l) + |a_{n-k} - \tilde{a}_{n-k}|, \end{aligned}$$

i.e.

$$\beta_{n-k} := \left(\beta_{n-(k-1)} + 2.1 \cdot \hat{T}_{n-(k-1)}\right) \cdot \hat{X} + (1 + \mathrm{ulp}(A_{n-k})) \cdot \hat{A}_{n-k}. \tag{C.7}$$

This estimation holds for any $x \in X$, that means especially for $\tilde{x}$.To find the bound β_{n-k} for the actual step an update using the bound already known from the preceding step has to be performed according to (C.7).
It is clear by construction that the quantities $\alpha := \alpha_0$ and $\beta := \beta_0$ will satisfy inequality (C.2).

We now want to use the methodology just described to find bounds α and β with

$$\begin{aligned}\left|\frac{xP_n(x) - \tilde{x} \odot \tilde{P}_n(\tilde{x})}{xP_n(x)}\right| \\ \leq \ldots \leq\end{aligned} \tag{C.8}$$

C.2 Program Code Using Taylor polynomials of $\exp(x)$

The program listed now computes enclosures for the coefficients of Taylor polynomials for $\exp(x)$ The enclosures are stored in a file called `out`. This file is used later on as input to the program `experr`.

```
{-----------------------------------------------------}
{ Generating enclosures of the coefficients           }
{                                                     }
{              1                                      }
{     a :=  -----                                     }
{      k      k!                                      }
{                                                     }
{ for the polynomial approximation of exp(x) using    }
{                                                     }
{               OO      k                             }
{    exp(x) = sum   a   x                             }
{             k=0    k                                }
{                                                     }
{-----------------------------------------------------}

PROGRAM ExpCoeff;

USE i_ari;
CONST nMax = 20;      { Maximal degree of the approximation }

VAR Coeff: ARRAY[0..Nmax] OF interval;   { Enclosures of ak }
    Out  : FILE OF interval;  { Communication file         }
    k    : integer;                       { Loop variable    }

BEGIN
   rewrite(Out, 'ExpCoeff.out');
   { The enclosures are stored for further use in file ExpCoeff.out }
   Coeff[0]:= 1;
   Coeff[1]:= 1;
   FOR k:= 2 TO nMax DO Coeff[k]:= Coeff[k-1]*k;     { Coeff[k] = k!   }

   FOR k:= 0 TO nMax DO Coeff[k]:= 1/Coeff[k];       { Coeff[k] = 1/k! }
   FOR k:= 0 TO nMax DO BEGIN
      { Write actual coefficient and information whether it is a point }
      {   interval to terminal.                                        }
      writeln(k:3, '  ',  Coeff[k], '  ', inf(Coeff[k])=sup(Coeff[k]) );
      Out↑:= Coeff[k];              { Write actual coefficient to file }
      put(Out);
      { Give a warning if the enclosure contains more than two }
      {    floating-point numbers.                             }
      {    In such a case, a multi-precision arithmetic should }
      {    be used.                                            }
      IF succ(inf(Coeff[k])) < sup(Coeff[k]) THEN
```

```
        writeln('W A R N I N G: Enclosure is not of maximum accuracy!');
    END;
END.
```

The following program is an implementation of the algorithm discussed in the previous section for the reliable estimation of rounding errors due to the machine evaluation of Horner's scheme. Here the algorithm is applied to Taylor polynomials for $\exp(x)$ in the basic range $[0, 0.125]$.

```
{----------------------------------------------------------------------}
{ Computation of the quantities alpha and beta according to            }
{ the error estimations given in the text.                             }
{ It holds:                                                            }
{   | p(x) - machine( p(machine(x)) ) |                                }
{            < alpha*relerr(x) + beta*epsilon                          }
{ Here epsilon means Wilkinson's epsilon 2**(-53) end relerr(x)        }
{ means a relative error bound for the machine equivalent to x.        }
{----------------------------------------------------------------------}
PROGRAM ExpErr;
USE i_ari;

CONST nMax = 20;   { Upper bound for the degree of the approximation. }

VAR
  coeff: ARRAY[0..nMax] OF interval;        { Enclosure of the  Coeff. }
  t: ARRAY[0..nMax] OF interval;      { Intermediate steps of Horner's }
  {                                                            scheme. }
  NumberOfParts: integer;                     { Number of subintervals. }
  xOrig: interval;                            { Reduced argument range. }
  x: interval;                                           { Subinterval. }
  alpha: real;       { Relative error bound with respect to relerr(x). }
  beta: real;          { Relative error bound with respect to epsilon. }
  h: real;                                 { Width of each subinterval. }
  AlphaMax: real;    { Maximal rel. error bound over all subintervals. }
  BetaMax: real;     { Maximal rel. error bound over all subintervals. }
  n: integer;                         { Actual degree of approximation. }
  k: integer;                                         { Loop variable. }
  error: boolean;                                        { Error code. }
          {  error = true if relative bounds can  n o t  be computed. }
          {  Their computation would cause a divide by zero!          }
  Inp: FILE OF interval;       { Enclosure of polynomial coefficients. }


FUNCTION max(x, y: real): real;
BEGIN IF x >= y THEN max:= x ELSE max:= y END;

FUNCTION MaxAbs(x: interval): real;           { max(|x.inf|, |x.sup|) }
BEGIN
  IF abs(inf(x)) >= abs(sup(x)) THEN
    MaxAbs:= abs(inf(x))
  ELSE
    MaxAbs:= abs(sup(x));
END;

FUNCTION Ulp( a: interval ): real;
{ The function value is 0 if a is a point interval and 1 if not. }
BEGIN
  IF inf(a) = sup(a) THEN Ulp := 0 ELSE Ulp := 1;
END;


BEGIN
```

```
    reset(Inp, 'ExpCoeff.out');
    { Enclosures of maximum accuracy for the coefficients of the        }
    {    Taylor polynomial for the exponential are read from file.      }
    xOrig:= intval(0, succ(succ(ln(2.0))) );      { Superset of [0, ln2] }
    writeln('(Reduced) Argument range: ', xOrig);
    write('Degree ?  '); read(n); writeln;
    writeln('Degree of polynomial: ', n);  { 0<=n<=nMax is not checked. }
    FOR k:= 0 TO n DO BEGIN
      coeff[k]:= Inp↑;      { Read enclosures of polynomial coeffients. }
      get(Inp);
      writeln('coeff[', k:2, ']: ', coeff[k]);
    END;
    AlphaMax:= 0;                 { rel. error bound valid for all parts }
    BetaMax := 0;
    error:= false;         { 0 is not an element of any result interval }

    { xOrig is subdivided into NumberOfParts subintervals. }
    NumberOfParts:= 975;
    x:= intval( inf(xOrig) );
    h:= (sup(xOrig) -> inf(xOrig)) /> NumberOfParts;

    REPEAT
      x:= intval(x.sup, x.sup + h);      { Generate actual subinterval. }
      IF sup(x) > sup(xOrig) THEN x.sup:= xOrig.sup;

      { Interval Horner's scheme for the actual subinterval: }
      t[n]:= coeff[n];
      FOR k:= 1 TO n DO BEGIN
        t[n-k]:= t[n-(k-1)]*x + coeff[n-k];
      END;
      IF 0 IN t[0] THEN error:= true;  { 0 in result, no relative error }
                                       {    bound is computable.        }
      { Quantity alpha in the actual subinterval: }
      alpha:= 0;
      FOR k:= 1 TO n DO BEGIN
        alpha:= alpha +> MaxAbs(t[n-(k-1)]) *> MaxAbs(X);
      END;
      AlphaMax:= max(alpha/inf(t[0]), AlphaMax);  { Relative error }
                                                  {    bound.      }
      { Quantity beta in the actual subinterval }
      beta:= ulp(coeff[n])*MaxAbs(coeff[n]);
      FOR k:= 1 TO n DO BEGIN
        beta:= ( beta +> 2.01*>MaxAbs(t[n-(k-1)]) )*MaxAbs(x)
                      +> (1 +> ulp(coeff[n-k]))*>MaxAbs(coeff[n-k]);
      END;
      BetaMax:= max(beta/inf(t[0]), BetaMax);     { Relative error }
                                                  {    bound.      }
    UNTIL x.sup = xOrig.sup;          { Loop over all subintervals }

    writeln('AlphaMax: ', AlphaMax:10:6);    { Bounds are valid for all }
    writeln('BetaMax : ', BetaMax :10:6);    {   subintervalls.         }
    IF error THEN
      writeln('W A R N I N G: Relative error bounds are  n o t  ok!')
  END.
  {-------------------------------------------------------------------}
```

Output:

```
(Reduced) Argument range: [ 0.0E+000,  7.0E-001 ]
Degree of polynomial: 15
coeff[ 0]: [  1.000000000000000E+000,  1.000000000000000E+000 ]
coeff[ 1]: [  1.000000000000000E+000,  1.000000000000000E+000 ]
```

```
coeff[ 2]: [  5.000000000000000E-001,  5.000000000000000E-001 ]
coeff[ 3]: [  1.666666666666666E-001,  1.666666666666667E-001 ]
   ...              ...                         ...
coeff[14]: [  1.147074559772972E-011,  1.147074559772973E-011 ]
coeff[15]: [  7.647163731819816E-013,  7.647163731819818E-013 ]
AlphaMax:   0.811837
BetaMax :   2.428265
```

Appendix D

Modul fnc_util

The module `fnc_util` provides procedures and functions which are used by the routines in Chapter ??. The routines are short and easy to read. The description relating to them is given as comments in the listing. The following routines are implemented: `Min`, `Max` for reals and intervals, `MaxAbs`, `Floor`, `IsEven`, `IsInteger`, `Pow2`, `Factorial`, `MaxRelErr` and the function `NextAfter`.

```
{----------------------------------------------------------------------}
{ Some support for function testing                                    }
{----------------------------------------------------------------------}
MODULE fnc_util;

USE i_ari;   { Interval operations }

{----------------------------------------------------------------------}
{ Function Min delivers the minimum of two real numbers.               }
{----------------------------------------------------------------------}
GLOBAL FUNCTION Min(r, s: real): real;
BEGIN
  IF r <= s THEN Min:= r ELSE Min:= s;
END;

{----------------------------------------------------------------------}
{ Function Max delivers the maximum of two real numbers.               }
{----------------------------------------------------------------------}
GLOBAL FUNCTION Max(r, s: real): real;
BEGIN
  IF r >= s THEN Max:= r ELSE Max:= s;
END;

{----------------------------------------------------------------------}
{ Function Min delivers the minimum of two real intervals.             }
{----------------------------------------------------------------------}
GLOBAL FUNCTION Min(x, y: interval): real;
BEGIN
  IF inf(x) <= inf(y) THEN Min:= inf(x) ELSE Min:= inf(y);
END;

{----------------------------------------------------------------------}
{ Function Max delivers the maximum of two real intervals.             }
{----------------------------------------------------------------------}
GLOBAL FUNCTION Max(x, y: interval): real;
BEGIN
  IF sup(x) >= sup(y) THEN Max:= sup(x) ELSE Max:= sup(y);
END;

{----------------------------------------------------------------------}
{ Function MaxAbs delivers the maximum of the absolute values          }
{ of its two arguments x and y, i.e.                                   }
{                MaxAbs(a, b) := max(|a|, |b|)                         }
```

```
{--------------------------------------------------------------------}
GLOBAL FUNCTION MaxAbs(a, b: real): real;
BEGIN
  IF abs(a) > abs(b) THEN MaxAbs:= abs(a) ELSE MaxAbs:= abs(b);
END;

{--------------------------------------------------------------------}
{ Function Floor delivers the largest integer smaller than or        }
{ equal to the argument x. Whether x is too large in magnitude       }
{ is not checked by this routine.                                    }
{--------------------------------------------------------------------}
GLOBAL FUNCTION Floor(x: real): integer;
VAR h: integer;
BEGIN
  h:= trunc(x);
  IF h > x THEN h:= h - 1;   { to work correct also if x is negative }
  Floor:= h;
END;

{--------------------------------------------------------------------}
{ Function IsInteger delivers true if its real argumnet x is an      }
{ integral quantity.                                                 }
{--------------------------------------------------------------------}
GLOBAL FUNCTION IsInteger(x: real): boolean;
BEGIN
  IF abs(x) > maxint THEN
    IsInteger:= false
  ELSE
    IsInteger:= 1.0*trunc(x) = x;
END;

{--------------------------------------------------------------------}
{ Function IsEven delivers true if its real argumnet x is an even    }
{ integer value.                                                     }
{--------------------------------------------------------------------}
GLOBAL FUNCTION IsEven(x: real): boolean;
BEGIN
  IF abs(x) > maxint THEN
    writeln('*** ERROR in IsEven: Argument too large!');
  IsEven:= NOT odd(trunc(x));
END;

{--------------------------------------------------------------------}
{ Function Pow2(n) delivers 2**n                                     }
{--------------------------------------------------------------------}
GLOBAL FUNCTION Pow2(n: integer): real;
  FUNCTION ldexp(x: real; n: integer): real; EXTERNAL 'ldexp';
  { ldexp is a function from the PASCAL-XSC runtime system. It is }
  { included here by an external statement. It is local to Pow2.  }
BEGIN
  IF n = 0 THEN
    Pow2:= 1
  ELSE IF n > 0 THEN
    Pow2:= ldexp(1.0, n)
  ELSE
    Pow2:= 1.0/ldexp(1.0, -n);
END;


{--------------------------------------------------------------------}
{ Function Factorial(n) delivers n*(n-1)*(n-2)* ...*2*1              }
{--------------------------------------------------------------------}
GLOBAL FUNCTION Factorial(n: integer): real;
VAR k: integer;
```

```
   r: real;
BEGIN
  r:= n;
  FOR k:= n-1 DOWNTO 2 DO r:= r*k;
  Factorial:= r;
END;

{------------------------------------------------------------------}
{ Compute an upper bound for the maximum relative error of an      }
{    interval enclosure y with respect to a second enclosure x     }
{    of some exact value:                                          }
{                                                                  }
{                       a - b                                      }
{      MaxRelErr >= | ------ |  for all a in x and all b in y      }
{                         a                                        }
{------------------------------------------------------------------}
GLOBAL FUNCTION MaxRelErr(x, y: interval): real;
{ It is assumed, that the correct value lies at least    }
{   in the interval x which appears in the denominator.  }
BEGIN
  IF x = 0 THEN BEGIN
    IF y = 0 THEN
      MaxRelErr:= 0
    ELSE
      writeln('*** MaxRelErr: Relative error possibly unbounded! ');
  END
  ELSE BEGIN
    MaxRelErr:= sup( abs((x-y)/x) );
  END;
END;

{------------------------------------------------------------------}
{ Compute floating-point numbers in the neighborhood of the        }
{    given argument x.                                             }
{ If the second argument n is positive, the n-th floating-point    }
{ number to the right of x is  returned, for n <= 0 the n-th flp   }
{ number to the left is returned.                                  }
{------------------------------------------------------------------}
GLOBAL FUNCTION nextafter(x: real; n: integer): real;
VAR
   r: real;
   i: integer;
BEGIN
  { If n is zero the value x itself is returned }
  r:= x;
  IF (n >= 0) THEN  { goto the right }
    FOR i:= 1 TO n DO r:= succ(r)
  ELSE              { goto the left  }
    FOR i:= 1 TO abs(n) DO r:= pred(r);
  nextafter:= r;
END;


{------------------------------------------------------------------}
{ Module initialization part                                       }
{------------------------------------------------------------------}
BEGIN
  { Nothing to initialize }
END.
```

Appendix E

Mathematical Appendix

see Toolbox I

Bibliography

[1] Aberth, O.: *Precise Numerical Analysis*. Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[2] Abramowitz, M., Stegun, I. A.: *Handbook of Mathematical Functions*. Nat. Bur. Standards, Appl. Math. Series, No. **55**, U.S. Government Printing Office, Washington, D.C. 1964.

[3] Adams, E., Lohner, R.: *Error Bounds and Sensitivity Analysis*. In Stepleman, R. S. (Ed.): *Scientific Computing*. North Holland, Amsterdam, 1983.

[4] Adams, E.: *Enclosure Methods and Scientific Computation*. In [23], pp 3–31, 1989.

[5] Adams, E., Kulisch, U. (Eds.): *Scientific Computing with Automatic Result Verification*. I. Language and Programming Support for Verified Scientific Computation, II. Enclosure Methods and Algorithms with Automatic Result Verification, III. Applications in the Engineering Sciences. Academic Press, San Diego, 1993.

[6] Adams, E., Kulisch, U.: *On Scientific Computing with Automatic Result Verification*. In [5], pp 1–12, 1993.

[7] Agarwal, R. C. et al.: *New Scalar and Vector Elementary Functions for the IBM System/370*. IBM J. Res. Develop., Vol. **30**, No. 2, 1986.

[8] Albrecht, R., Kulisch, U. (Eds.): *Grundlagen der Computerarithmetik*. Computing Supplementum **1**, Springer-Verlag, Wien, New York, 1977.

[9] Albrecht, R., Alefeld, G., Stetter, H. J. (Eds.): *Validation Numerics – Theory and Applications*. Computing Supplementum **9**, Springer-Verlag, Wien / New York, 1993.

[10] Alefeld, G., Herzberger, J.: *Einführung in die Intervallrechnung*. Bibliographisches Institut (Reihe Informatik, No. 12), Mannheim / Wien / Zürich, 1974.

[11] Alefeld, G.: *Über die Durchführbarkeit des Gaußschen Algorithmus bei Gleichungen mit Intervallen als Koeffizienten*. In [8], pp 15–19, 1977.

[12] Alefeld, G., Grigorieff, R. D. (Eds.): *Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis)*. Computing Supplementum **2**, Springer-Verlag, Wien / New York, 1980.

[13] Alefeld, G., Herzberger, J.: *Introduction to Interval Computations*. Academic Press, New York, 1983.

[14] Alefeld, G., Lohner, R.: *On Higher Order Centered Forms*. Computing **35**, pp 177–184, 1985.

[15] Alefeld, G.: *Errorbounds for Quadratic Systems of Nonlinear Equations Using the Precise Scalar Product*. In [216], pp 59–67, 1988.

[16] Alefeld, G.: *Existence of Solutions and Iterations for Nonlinear Equations*. In [249], pp 207–227, 1988.

[17] Alefeld, G.: *Enclosure Methods*. In [322], pp 55–72, 1990.

[18] Alefeld, G., Illg, B., Potra, F.: *On a Class of Enclosure Methods for Systems of Equations with Higher Order of Convergence*. In [322], pp 151–159, 1990.

[19] Alefeld, G., Mayer, G.: *The Cholesky Method for Interval Data*. Presentation at the International Conference on "Numerical Analysis with Automatic Result Verification". Lafayette, Louisiana, USA, Feb. 25 – March 1, 1993.

[20] Allendörfer, U., Shiriaev, D.: *PASCAL–XSC to C – A Portable PASCAL–XSC Compiler*. In [164], pp 91–104, 1991.

[21] Allendörfer, U., Shiriaev, D.: *PASCAL–XSC. A Portable Development System*. Proceedings of 13th World Congress on Computation and Applied Mathematics, IMACS '91, Dublin, 1991.

[22] Allendörfer, U., Shiriaev, D.: *PASCAL–XSC. A Portable Development System*. In [76], 1992.

[23] Ames, W. F. (Ed.): *Numerical and Applied Mathematics*. J. C. Baltzer Scientific Publishing, Basel, 1989.

[24] Andreev, A. S., Dimov, I. T., Markov, S. M., Ullrich, Ch. (Eds.): *Mathematical Modelling and Scientific Computations*. Bulgarian Academy of Sciences, Sofia, 1991.

[25] ANSI/IEEE Standard 754-1985, *Standard for Binary Floating-Point Arithmetic*. New York, 1985.

[26] Atanassova, L., Herzberger, J. (Eds.): *Computer Arithmetic and Enclosure Methods*. Proceedings of SCAN 91, North-Holland, Elsevier Science Publishers B.V., Amsterdam, 1992.

[27] Auzinger, W., Stetter, H. J.: *Accurate Arithmetic Results for Decimal Data on Non-Decimal Computers*. Computing **35**, pp 141–151, 1985.

[28] Bailey, D. H.: *A Portable High Performance Multiprecision Package*. RNR Technical Report RNR-90-022, 1993.

[29] Bailey, D. H.: *A Fortran-90 Based Multiprecision System*. RNR Technical Report RNR-94-013, 1995.

[30] Barth, B.: *Eine verifizierte Einschließung von Werten der Weierstraßschen $\wp$-Funktion*. Diplomarbeit am Institut für Angewandte Mathematik, Universität Karlsruhe, 1991.

[31] Bauch, H., Jahn, K.-U., Oelschlägel, D., Süsse, H., Wiebigke, V.: *Intervallmathematik, Theorie und Anwendungen*. BSB B.G. Teubner Verlagsgesellschaft, Leipzig, 1987.

[32] Kernhof, J.; Baumhof, Ch.; Höfflinger, B.; Kulisch, U.; Kwee, S.; Schramm, P.; Selzer, M.; Teufel, Th.: *A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic.* ESSIRC 94, Ulm, Sept. 1994.

[33] Baumhof, Ch.: *A New VLSI Vector Arithmetic Coprocessor for the PC.* To appear in: Proceedings of the 12th Symposium on Computer Arithmetic, IEEE Computer Society, 1995.

[34] Behnke, H.: *Inclusion of Eigenvalues of General Eigenvalue Problems of Matrices*. In [216], pp 69–78, 1988.

[35] Behnke, H.: *The Determination of Guaranteed Bounds to Eigenvalues with the Use of Variational Methods* **II**. (Part **I** see [112]). In [322], pp 155–170, 1990.

[36] Behnke, H.: *The Calculation of Guaranteed Bounds for Eigenvalues Using Complementary Variational Principles*. Computing **47**, pp 11–27, 1992.

[37] Berz, M.: *Automatic Differentiation as Nonarchimedean Analysis*. In [26], pp 439–450, 1992.

[38] Bethke, D., Herzberger, J.: *Über Eigenschaften von zwei Methoden zur Einschließung der Inversen einer Intervallmatrix*. In [151], pp 409–413, 1991.

[39] Bischof, C., Carle, A., Corliss, G., Griewank, A., Hovland, P.: *ADIFOR: Fortran Source Translation for Efficient Derivatives*. ADIFOR Working Note No. **4**. Argonne National Laboratory, address: see [90]. Report MCS-P278-1291, February 1992.

[40] Black, Ch. M., Burton, R. B., Miller, T. H.: *The Need for an Industry Standard of Accuracy for Elementary-Function Programs*. ACM Trans. on Math. Software, Vol. **10**, No. 4, pp 361–366, 1984.

[41] Bleher, J. H., Rump, S. M., Kulisch, U., Metzger, M., Ullrich, Ch., Walter, W.: *FORTRAN-SC: A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH*. Computing **39**, pp 93–110, 1987. Also in [216], pp 227–244, 1988.

[42] Bleher, J. H., Rump, S. M., Kulisch, U., Metzger, M., Ullrich, Ch., Walter, W.: *FORTRAN–SC: A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH*. In [216], pp 227–244, 1988.

[43] Blomquist, F.: *Implementierung und Fehlerabschätzungen von PASCAL-XSC Standardfunktionen für ein dezimales Datenformat*. Universität Karlsruhe, 1992.

[44] Blomquist, F.: *Implementierung einiger spezieller mathematischer Funktionen in PASCAL-XSC*. Private Mitteilung, 1992.

[45] Bochev, P., Markov, Sv.: *A Self-Validating Numerical Method for the Matrix Exponential*. Computing **43**, pp 59–72, 1989.

[46] Bochev, P., Markov, S.: *Simultaneous Self-Verified Computation of $exp(A)$ and $\int_0^1 exp(As)ds$*. Computing **45**, pp 183–191, 1990.

[47] Bohlender, G.: *Genaue Summation von Gleitkommazahlen*. In [8], pp 21–32, 1977.

[48] Bohlender, G., Grüner, K., Kaucher, E., Klatte, R., Krämer, W., Kulisch, U., Rump, S., Ullrich, Ch., Wolff von Gudenberg, J., Miranker, W.: *PASCAL-SC: A PASCAL for Contemporary Scientific Computation*. Research Report RC 9009, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1981.

[49] Bohlender, G., Rall, L., Ullrich, Ch. und Wolff von Gudenberg, J.: *PASCAL-SC – Wirkungsvoll programmieren, kontrolliert rechnen*. Bibliographisches Institut, Mannheim, 1986.

[50] Bohlender, G., Teufel, T.: *Demonstration of the Bit-Slice Processor Unit BAP–SC in a 68000 Environment*. In [328],Vol. **1**, pp 155–158, [293], pp 331–336, 1986.

[51] Bohlender, G., Teufel, T.: *BAP–SC: A Decimal Floating-Point Processor for Optimal Arithmetic*. In [163], pp 31–58, 1987.

[52] Bohlender, G., Rall, L., Ullrich, Ch., Wolff von Gudenberg, J.: *PASCAL-SC: A Computer Language for Scientific Computation*. Academic Press, New York, 1987.

[53] Bohlender, G., Wolff von Gudenberg, J., Miranker, W. L.: *Floating-Point Systems for Theorem Proving*. In Meyer, K. R., Schmidt D. S.: *Computer Aided Proofs in Analysis*. Springer Verlag, 1990.

[54] Bohlender, G.: *A Vector Extension of the IEEE Standard for Floating-Point Arithmetic*. In [164], pp 3–12, 1991.

[55] Bohlender, G., Knöfel, A.: *A Survey of Pipelined Hardware Support for Accurate Scalar Products*. In [164], pp 29–43, 1991.

[56] Bohlender, G., Cordes, D., Knöfel, A., Kulisch, U., Lohner, R., Walter, W. V.: *Proposal for Accurate Floating-Point Vector Arithmetic*. In [5], pp 87–102, 1993.

[57] Bohlender, G.: *Bibliography on Enclosure Methods and Related Topics*. In [5], pp 571–608, 1993.

[58] Bohlender, G., Krämer, W., Miranker, W. L.: Grading of Basic Arithmetical Operations and Functions. IBM Rsearch Report, RC 19593(86059)6—1—94, 1994.

[59] Böhm, H.: *Auswertung arithmetischer Ausdrücke mit maximaler Genauigkeit*. In [212], pp 175–184, 1982.

[60] Böhm, H.: *Berechnung von Polynomnullstellen und Auswertung arithmetischer Ausdrücke mit garantierter maximaler Genauigkeit.* Dissertation, Universität Karlsruhe, 1983.

[61] Böhm, H.: *Evaluation of Arithmetic Expressions with Maximum Accuracy.* In [213], pp 121–137, 1983.

[62] Borwein, J. M., Borwein, P. B.: *The Arithmetic-Geometric Mean and Fast Computation of Elementary Functions.* SIAM Review, Vol. **26**, No. 3, July 1984.

[63] Borwein, J. M., Borwein, P. B.: *Pi and the AGM.* John Wiley & Sons, 1987.

[64] Borwein, J. M., Borwein, P. B., Bailey, D. H.: *Ramanujan, Modular Equations, and Approximations to Pi.* Amer. Math. Monthly 96, pp 201–219, 1989.

[65] Brassard, G., Monet, S., Zuffellato, D.: *Algorithmes pour l'arithmétique des très grands entiers.* TSI Technique et Science Informatiques, Vol. **5**, No. 2, 1986.

[66] Braune, K., Krämer, W.: *Standard Functions for Intervals with Maximum Accuracy.* 11th IMACS World Congress, Proceedings Vol. **1**, pp 167–170, 1985.

[67] Braune, K., Krämer, W.: *High-Accuracy Standard Functions for Intervals.* In M. Ruschitzka (Ed.): *Computer Systems: Performance and Simulation.* Elsevier Science Publishers, 1985.

[68] Braune, K., Krämer, W.: *High Accuracy Standard Functions for Real and Complex Intervals.* In Kaucher, E., Kulisch, U., Ullrich, Ch: *Computerarithmetic: Scientific Computation and Programming Languages*, pp81–114, Teubner, Stuttgart, 1987.

[69] Braune, K.: *Hochgenaue Standardfunktionen für reelle und komplexe Punkte und Intervalle in beliebigen Gleitpunktrastern.* Dissertation, Universität Karlsruhe, 1987.

[70] Braune, K.: *Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy.* Computing Supplement **6**, pp 227–244, 1988.

[71] Braune, K.: *A-posteriori Fehlerschranken bei der Berechnung inverser Standardfunktionen mit Hilfe des Newton-Verfahrens.* ZAMM **70**, pp T579–T581, 1990.

[72] Brent, P. B.: *Fast Multiple-Precision Evaluation of Elementary Functions.* J. of the Association for Computing Machinery, Vol. **23**, No. 2, 242-251, 1976.

[73] Brent, R. P.: *A FORTRAN Multiple Precision Arithmetic Package.* ACM Trans. Math. Software **4**, pp 57–70, 1978.

[74] Brent, R. P., Hooper, J. A., Yohe, J. M.: *An Augment Interface for Brent's Multiple-Precision Arithmetic Package.* ACM Trans. Math. Software **6**, pp 146–149, 1980.

[75] Brent, R. P.: *MP User's Guide*. 4th ed., Technical Report TR–CS–81–08, Department of Computer Science, Australian National University, Canberra, 1981.

[76] Brezinski, C., Kulisch, U. (Eds.): *Computational and Applied Mathematics – Algorithms and Theory*. Proceedings of the 13th IMACS World Congress, Dublin, Ireland. Elsevier, Science Publishers B.V., 1992.

[77] Bulirsch, R., Stoer, J.: *Asymptotic Upper and Lower Bounds for Results of Extrapolation Methods*. Numerische Mathematik **8**, pp 93–104, 1966.

[78] Cordes, D., Kaucher, E.: *Self-Validating Computation for Sparse Matrix Problems*. In [163], pp 133–149, 1987.

[79] Cordes, D.: *Spärlich besetzte Matrizen*. In [217], pp 129–136, 1989.

[80] Cordes, D., Krämer, W.: *Vom Problem zum Einschließungsalgorithmus*. In Kulisch, U.: *Wissenschaftliches Rechnen mit Ergebnisverifikation*, pp 167–181, Vieweg, Braunschweig, 1989.

[81] Cordes, D.: *PASCAL-XSC Runtime Library*. SCAN 90, 1990.

[82] Cordes, D.: *Runtime System for a PASCAL–XSC Compiler*. In [164], pp 151–160, 1991.

[83] Cordes, D., Krämer, W.: *PASCAL-XSC Module for Multiple-Precision Operations and Functions*. Universität Karlsruhe, 1991.

[84] Corliss, G. F., Rall, L. B.: *Adaptive, Self-Validating Numerical Quadrature*. MRC Technical Summary Report # 2815, University of Wisconsin, Madison, 1985.

[85] Corliss, G. F.: *Computing Narrow Inclusions for Definite Integrals*. In [163], pp 150–169, 1987.

[86] Corliss, G. F.: *Applications of Differentiation Arithmetic*. In [249], pp 127–148, 1988.

[87] Corliss, G. F.: *Automatic Differentiation Bibliography*. In [115], pp 331–353, 1991. See also [90].

[88] Corliss, G. F., Rall, L. B.: *Computing the Range of Derivatives*. In [164], pp 195–212, 1991.

[89] Corliss, G. F.: *Validated Anti-Derivatives*. In Meyer, R. K., Schmidt, D. S.: *Computer Aided Proofs in Analysis*. Springer-Verlag (The IMA Volumes in Mathematics and Its Applications, Vol. **28**), New York, 1991.

[90] Corliss, G. F. (Ed.): *Automatic Differentiation Bibliography*. Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, Illinois, Report ANL/MCS-TM-167 (30 pages), July 1992.

[91] Cornelius, H., Lohner, R.: *Computing the Range of Values of Real Functions with Accuracy Higher Than Second Order*. Computing **33**, pp 331–347, 1984.

[92] Cornelius, H., Lohner, R.: *Enclosing the Range of Values of Real Functions.* ZAMM **64**, pp T408–T410, 1984.

[93] Cyrix: *FasMath CX-83D87 User's Manual.* Cyrix Corporation, P.O. Box 850118, Richardson, TX 75085-0118, 1990.

[94] Davis, Ph. J., Rabinowitz, Ph.: *Methods of Numerical Integration.* Academic Press, San Diego, 1984.

[95] Dekker, T. J.: *A Floating-Point Technique for Extending the Available Precision.* Numische Mathematik **18**, pp 224–242, 1971.

[96] Egbert, W. E.: *Personal Calculator Algorithms I, II, III and IV.* Hewlett-Packard Journal, Mai 1977, Juni 1977, November 1977 und April 1978.

[97] Engels, H.: *Numerical Quadrature and Cubature.* Academic Press, New York, 1980.

[98] Erb, H.: *Ein Gleitpunkt-Arithmetikprozessor mit mehrfacher Präzision zur verifizierten Lösung linearer Gleichungssysteme.* Dissertation, Fakultät für Informatik, Universität Karlsruhe, 1992.

[99] Fernando, K. V., Pont, M. W.: *Computing Accurate Eigenvalues of a Hermitian Matrix.* In [321], pp 104–115, 1989.

[100] Fischer, H.: *Fast Method to Compute the Scalar Product of Gradient and Given Vector.* Computing **41**, pp 261–265, 1986.

[101] Fischer, H.-C.: *Schnelle automatische Differentiation, Einschliessungsmethoden und Anwendungen.* Dissertation, Universität Karlsruhe, 1990.

[102] Fischer, H.-C.: *Range Computation and Applications.* In [322], pp 197–211, 1990.

[103] Fischer, H.-C.: *Effiziente Berechnung von Ableitungswerten, Gradienten und Taylorkoeffizienten.* In Chatterji, S. D., Fuchssteiner, B., Kulisch, U., Liedl, R., Purkert, W. (Eds.): *Jahrbuch Überblicke Mathematik 1992*, Vieweg, Braunschweig, 1992.

[104] Fischer, H.-C.: *Automatic Differentiation and Applications.* In [5], pp 105–142, 1993.

[105] Frommer, A.: *Lösung linearer Gleichungssysteme auf Parallelrechnern.* Vieweg Verlag, Braunschweig, 1990.

[106] Frommer, A.: *Asynchronous Iterations for Enclosing Solutions of Fixed Point Problems.* In [26], pp 243–252, 1992.

[107] Gaffney, P., Houstis, E. (Eds.): *Programming Environments for High Level Scientific Problem Solving.* Proceedings of IFIP WG 2.5 conference, Karlsruhe, September 23–27, 1991, Elsevier Science Publishers, 1993.

[108] Gal, S.: *Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance.* IBM Technical Report 88.153, 1985.

[109] Gal, S., Bachelis, B.: *An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard.* IBM Technical Report 88.223, IBM Israel, Technion City, Haifa, Israel, 1988.

[110] Garloff, J.: *Interval Mathematics. A Bibliography.* Freib. Int.-Ber. **6**, pp 1–222, 1985

[111] Garloff, J.: *Bibliography on Interval Mathematics. Continuation.* Freib. Int.-Ber. **2**, pp 1–50, 1987.

[112] Goerisch, F., He, Z.: *The Determination of Guaranteed Bounds to Eigenvalues with the Use of Variational Methods* **I**. (Part **II** see [35]). In [322], pp 137–153, 1990.

[113] Gregory, R. T., Karney, D. L.: *A Collection of Matrices for Testing Computational Algorithms.* John Wiley & Sons, New York, 1969.

[114] Griewank, A.: *On Automatic Differentiation.* In Iri, M., Tanabe, K. (Eds.): *Mathematical Programming: Recent Developments and Applications*, pp 83–108, Kluwer Academic Publishers, 1989.

[115] Griewank, A., Corliss, G. (Eds.): *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* Proceedings of Workshop on Automatic Differentiation at Breckenridge, SIAM, Philadelphia, 1991.

[116] Griewank, A.: *Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation.* Preprint MCS-P228-0491, Argonne National Laboratory, Argonne, 1991.

[117] Grüner, K.: *Fehlerschranken für lineare Gleichungssysteme.* In [8], pp 47–55, 1977.

[118] Grüner, K.: *Solving Complex Problems for Polynomials and Linear Systems with Verified High Accuracy.* In [163], pp 199–220, 1987.

[119] Grüner, K.: *Solving the Complex Eigenvalue Problem with Verified High Accuracy.* ESPRIT project DIAMOND, Doc. No. 03/3-2/1/Kl.p, 1987.

[120] Grüner, K.: *Solving the Complex Algebraic Eigenvalue Problem with Verified High Accuracy.* In [321], pp 59–78, 1989.

[121] Hammer, R.: *How Reliable is the Arithmetic of Vector Computers?* In [323], pp 467–482, 1990.

[122] Hammer, R.: *Maximal genaue Berechnung von Skalarproduktausdrücken und hochgenaue Auswertung von Programmteilen.* Dissertation, Universität Karlsruhe, 1992.

[123] Hammer, R.: *PASCAL–XSC: From Accurate Expressions to the Accurate Evaluation of Program Parts.* In [26], pp 119–128, 1992.

[124] Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: *Numerical Toolbox for Verified Computing I: Basic Numerical Problems.* Springer–Verlag, Berlin / Heidelberg / New York, 1993.

[125] Hammer, R., Neaga, M., Ratz, D.: *PASCAL–XSC, New Concepts for Scientific Computation and Numerical Data Processing.* In [5], pp 15–44, 1993.

[126] Hammer, R.: *Multi-Precision Arithmetic in PASCAL–XSC, Implementation and Applications.* Talk at SCAN-93, Wien, 1993.

[127] Hansen, E.: *Topics in Interval Analysis.* Clarendon Press, Oxford, 1969.

[128] Hart, J. F. et al.: *Computer Approximations.* Wiley, New York / London / Sydney, 1968.

[129] Heindl, G.: *Inclusions of the Range of Functions and their Derivatives.* In [164], pp 229–238, 1991.

[130] Herzberger, J.: *Intervallmäßige Auswertung von Standardfunktionen in ALGOL-60.* Computing **5**, pp 377–384, 1970.

[131] Herzberger, J.: *On Schulz's Method in Circular Complex Arithmetic.* In [323], pp 93–102, 1990.

[132] Herzberger, J., Petković, Lj.: *Efficient Iterative Algorithms for Bounding the Inverse of a Matrix.* Computing **44**, pp 237–244, 1990.

[133] Herzberger, J.: *Iterative Inclusion of the Inverse Matrix.* In [24], pp 14–26, 1991.

[134] Herzberger, J., Bethke, D.: *Interval Schulz's Method: On the Case of an Interval Matrix.* In [26], pp 199–203, 1992.

[135] Hofschuster, W., Krämer, W.: *Rechnergestütztes Fehlerkalkül mit Anwendung auf ein genaues Tabellenverfahren.* Preprint des IWRMM, Universität Karlsruhe, 1996.

[136] Hull, T. E., Abrham, A.: *Properly Rounded Variable Precision Square Root.* ACM Trans. on Math. Software, Vol. **11**, No. 3, pp 229–237, 1985.

[137] Hull, T. E., Abrham, A, Cohen, M. S., Curley, A. F. X., Penny, D. A., Sawchuk, J. T. M.: *Numerical Turing.* ACM SIGNUM Newsletter **20**, No. 3 pp 26–34, 1985.

[138] Husung, D.: *TPX Version 1.1 US Precompiler for Turbo Pascal 5.0.* Technische Universit"at Hamburg-Harburg, Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, Bericht 93.2, 1993.

[139] IBM: *IBM System/370 RPQ. High Accuracy Arithmetic.* SA 22-7093-0, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, 7030 Böblingen), 1984.

[140] IBM: *IBM High-Accuracy Arithmetic Subroutine Library (ACRITH).* IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, 7030 Böblingen), 3rd edition, 1986.
1. General Information Manual. GC 33-6163-02.
2. Program Description and User's Guide. SC 33-6164-02.
3. Reference Summary. GX 33-9009-02.

[141] IBM: *ACRITH–XSC: IBM High Accuracy Arithmetic — Extended Scientific Computation. Version 1, Release 1*. IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, 7030 Böblingen), 1990.
1. General Information, GC33-6461-01.
2. Reference, SC33-6462-00.
3. Sample Programs, SC33-6463-00.
4. How To Use, SC33-6464-00.
5. Syntax Diagrams, SC33-6466-00.

[142] American National Standards Institute / Institute of Electrical and Electronics Engineers: *IEEE Standard Pascal Computer Programming Language*. ANSI/IEEE Std. 770 X3.97-1983, New York, 1983; J. Wiley & Sons Inc., 1983.

[143] American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std. 754-1985, New York, 1985 (reprinted in SIGPLAN **22**, 2, pp 9–25, 1987).

[144] American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Radix-Independent Floating-Point Arithmetic*. ANSI/IEEE Std. 854-1987, New York, 1987.

[145] IMACS, GAMM: *IMACS-GAMM Resolution on Computer Arithmetic*. In *Mathematics and Computers in Simulation* **31**, pp 297–298, 1989. In *Zeitschrift für Angewandte Mathematik und Mechanik* **70**, No. 4, pp T5, 1990. In [322], pp 301–302, 1990. In [323], pp 523–524, 1990. In [164], pp 477–478, 1991.

[146] IMACS, GAMM: *IMACS-GAMM Proposal for Accurate Floating-Point Vector Arithmetic*. GAMM, Rundbrief **2**, pp 9–16, 1993. Mathematics and Computers in Simulation, Vol. **35**, IMACS, North Holland, 1993.

[147] Intel: *80287 Support Library Reference Manual*. Order Number 122669-001. Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051, 1985.

[148] Iri, M.: *Simultaneous Computation of Functions, Partial Derivatives and Estimates for Rounding Errors – Complexity and Practicality*. Japan Journal of Applied Mathematics **1**, pp 223–252, 1984.

[149] ISO/IEC: *ISO/IEC 7185:1990, Information Technology – Programming Languages – Pascal*. Second edition, 1990.

[150] ISO/IEC: *ISO/IEC 10206:1991 – Information Technology – Programming Languages – Extended Pascal*. 1991.

[151] Jahn, K.-U. (Ed.): *Computernumerik mit Ergebnisverifikation*. Problemseminar, Technische Hochschule Leipzig, 13.-15. März 1991. Proceedings in Wissenschaftliche Zeitschrift der Technischen Hochschule Leipzig, Jahrgang 15, Heft 6, 1991.

[152] Jansson, C.: *Guaranteed Error Bounds for the Solution of Linear Systems*. In [323], pp 103–110, 1990.

[153] Jansson, C.: *A Fast Direct Method for Computing Verified Inclusions*. Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, Technische Universität Hamburg-Harburg, Bericht 90.4, 1990.

[154] Jansson, C., Rump, S. M.: *Rigorous Sensitivity Analysis for Real Symmetric Matrices with Uncertain Data*. In [164], pp 293–316, 1991.

[155] Jansson, C.: *Interval Linear Systems with Symmetric Matrices, Skew-Symmetric Matrices and Dependencies in the Right Hand Side*. Computing **46**, pp 265–274, 1991.

[156] Jansson, C.: *A Geometric Approach for Computing A Posteriori Error Bounds for the Solution of a Linear System*. Computing **47**, pp 1–9, 1991.

[157] Jansson, C.: *A Posteriori Error Bounds for the Solution of Linear Systems with Simplex Techniques*. To appear 1992.

[158] Jensen, K., Wirth, N.: *Pascal User Manual and Report*. ISO Pascal Standard, 3rd ed., Springer, NewYork, 1985.

[159] Juedes, D. W.: *A Taxonomy of Automatic Differentiation Tools*. Argonne National Laboratory, address: see [90]. Report MCS-P265-0991, 1991.

[160] Kahan, W.: *Minimizing '$q \cdot m - n$'*. Unpublished research notes, March 1983.

[161] Kahan, W.: *Branch Cuts for Complex Elementary Functions*. In Iserles, A., Powell, M. J. D. (Eds.): *The state of the Art in Numerical Analysis. Clarendon Press,* Oxford, 1987.

[162] Karatsuba, A., Ofman, Y.: *Multiplication of Multidigit Numbers on Automata*. Soviet Physics Dokl. 7, pp 595–596, 1963.

[163] Kaucher, E., Kulisch, U., Ullrich, Ch. (Eds.): *Computerarithmetic: Scientific Computation and Programming Languages*. B. G. Teubner Verlag, Stuttgart, 1987

[164] Kaucher, E., Mayer, G., Markov, S. M. (Eds.): *Computer Arithmetic, Scientific Computation and Mathematical Modelling*. Proceedings of SCAN-90. IMACS Annals on Computing and Applied Mathematics, Vol. **12** (1992), published Oct. 1991. J. C. Baltzer AG, Basel, 1991.

[165] Kelch, R.: *Ein adaptives Verfahren zur numerischen Quadratur mit automatischer Ergebnisverifikation*. Dissertation, Universität Karlsruhe, 1989.

[166] Kelch, R.: *Self-Validating Numerical Quadrature*. In [321], pp 477–478, 1989.

[167] Kelch, R.: *An Adaptive Procedure for Numerical Quadrature with Automatic Result Verification*. In [323], pp 301–317, 1990.

[168] Kelch, R.: *Numerical Quadrature by Extrapolation with Automatic Result Verification*. In [5], pp 143–185, 1993.

[169] Kießling, I., Lowes, M., Paulik, A.: *Genaue Rechnerarithmetik — Intervallrechnung und Programmieren mit PASCAL–SC*. Teubner, Stuttgart, 1988.

[170] Kirchner, R., Kulisch, U.: *Schaltungsanordnung und Verfahren zur schnellen Berechnung von Summen und Skalarprodukten von Gleitkommazahlen mit maximaler Genauigkeit mittels Pipelinetechnik*. In Beckmann, M. J.; Gaede, K. W.; Ritter, K. (Eds.): *Beiträge zur Angewandten Mathematik und Statistik*, pp 139–177, Hansa–Verlag, München / Wien, 1987.

[171] Kirchner, R., Kulisch, U.: *Accurate Arithmetic for Vector Processors*. Journal of Parallel and Distributed Computing **5**, pp 250–270, 1988.

[172] Kirchner, R., Kulisch, U.: *Fast and Accurate Computation of Sums and Inner Products*. In Noye, J., Fletcher, C. (Eds.): *Computational Techniques and Applications: CTAC–87*, pp 3–28, Proceedings of CTAC–87 in Sidney, Australia. North–Holland, 1988.

[173] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ullrich, Ch.: *PASCAL–XSC — Sprachbeschreibung mit Beispielen*. Springer-Verlag, Berlin/Heidelberg/New York, 1991.

[174] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ullrich, Ch.: *PASCAL–XSC — Language Reference with Examples*. Springer-Verlag, Berlin/Heidelberg/New York, 1992.

[175] Klatte, R., Kulisch, U., Lawo, C., Rauch, M., Wiethoff, A.: *C–XSC, A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, Berlin/Heidelberg/New York, 1993.

[176] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ullrich, Ch.: *PASCAL–XSC — Language Reference with Examples (In Russian)*. MIR-Verlag, Moskau. To appear 1995.

[177] Klein, W.: *Verified Solution of Linear Systems with Band-Shaped Matrices*. DIAMOND Workpaper, Doc. No. 03/3-3/3, January 1987.

[178] Klein, W.: *Data Structure and Symbolic LU-Factorization for General Sparse Matrices*. DIAMOND Workpaper, Doc. No. 03/3-7/3, March 1987.

[179] Klein, W.: *Verified Results for Linear Systems with Sparse Matrices*. In [321], pp 137–161, 1989.

[180] Klotz, G.: *Faktorisierung von Matrizen mit maximaler Genauigkeit*. Dissertation, Universität Karlsruhe, 1987.

[181] Klug, U.: *Verified Inclusions for Eigenvalues and Eigenvectors of Real Symmetric Matrices*. In [323], pp 111–125, 1990.

[182] Knöfel, A.: *Hardwareentwurf eines Rechenwerks für semimorphe Skalar- und Vektoroperationen unter Berücksichtigung der Anforderungen verifizierender Algorithmen*. Dissertation, Universität Karlsruhe, 1991.

[183] Knöfel, A.: *Advanced Circuits for the Computation of Accurate Scalar Products*. In [164], pp 63–67, 1991.

[184] Knöfel, A.: *A Hardware Kernel for Scientific/Engineering Computations*. In [5], pp 549–570, 1993.

[185] Knuth, D. E.: *Euler's Constant to 1271 Places*. Math. Comp. **16**, pp 275–281, 1962.

[186] Knuth, D. E.: *The Art of Computer Programming, Volume* **2**/*Seminumerical Algorithms*. 2. Auflage, Addison-Wesley, Reading, 1981.

[187] König, S.: *On the Inflation Parameter Used in Self-validating Methods*. In [323], pp 127–132, 1990.

[188] Krämer, W.: *Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate*. Dissertation, Universität Karlsruhe, 1987

[189] Krämer, W.: *Inverse Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy*. Computing, Supplementum **6**, pp 185–212, 1988.

[190] Krämer, W.: *Mehrfachgenaue reelle und intervallmäßige Staggered-Correction Arithmetik mit zugehörigen Standardfunktionen*. Bericht des Instituts für Angewandte Mathematik, Universität Karlsruhe, pp 1–80, 1988.

[191] Krämer, W.: *Fehlerschranken für häufig auftretende Approximationsausdrücke*. ZAMM **69**, pp T44–T47, 1989.

[192] Krämer, W.: *Genaue Auswertung von Polynomen in mehreren Variablen*. DFG-Bericht zum Forschungsvorhaben Ku 155/12-1, 1989.

[193] Krämer, W.: *Berechnung der Gammafunktion $\Gamma(X)$ für reelle Punkt- und Intervallargumente*. ZAMM **70**, pp T581–T584,1990.

[194] Krämer, W.: *Highly Accurate Evaluations of Program Parts with Applications*. In C. Ullrich (Ed.): *Contributions to Computer Arithmetic and Self-Validating Numerical Methods*. J. C. Baltzer AG, Scientific Publishing Co., IMACS, 1991, pp 397–409, 1990.

[195] Krämer, W.: *Computation of Verified Bounds for Elliptic Integrals*. Proceedings of the International Symposium on Computer Arithmetic and Scientific Computation, Oldenburg 1991 (SCAN91). Edited by J. Herzberger and L. Atanassova; Elsevier Science Publishers (North-Holland).

[196] Krämer, W.: *Evaluation of Polynomials in Several Variables with High Accuracy*. In E. Kaucher, S. M. Markov, G. Mayer (Eds.): *Computer Arithmetic, Scientific Computation and Mathematical Modelling*, pp 239–249, I. C. Baltzer AG, Scientific Publishing Co., IMACS, 1991.

[197] Krämer, W.: *Einschluß eines Paares konjugiert komplexer Nullstellen eines reellen Polynoms*. ZAMM **71**, pp T820–T524, 1991.

[198] Krämer, W.: *Genaue Berechnung von komplexen Polynomen in mehreren Variablen*. In [151], pp 401–407, 1991.

[199] Krämer, W.: *Verified Solution of Eigenvalue Problems with Sparse Matrices*. In [326], pp 32–33, 1991.

[200] Krämer, W.: *PASCAL-XSC Moduless for Multiple-Precision Interval Operations and Functions*. Universität Karlsruhe, 1991.

[201] Krämer, W.: *Die Berechnung von Standardfunktionen in Rechenanlagen*. In Chatterji, S. D., Kulisch, U., Laugwitz, D., Liedl, R., Purkert, W. (Eds.): *Jahrbuch Überblicke Mathematik 1992*, Vieweg, Braunschweig, 1992.

[202] Krämer, W.: *Multiple-Precision Computations with Result Verification*. In [5], 1992.

[203] Krämer, W.: *Eine portable Langzahl- und Langzahlintervallarithmetik mit Anwendungen*. ZAMM **73**, 1992.

[204] Krämer, W.: *Verified Solution of Eigenvalue Problems with Sparse Matrices*. In Brezinsky, C. (Ed.): *Computational and Applied Mathematics I, Algorithms and Theory*, Proceedings of the 13th IMACS World Congress, Dublin, Ireland. Elsevier Science Publishers B. V., pp 1–11, 1992.

[205] Krämer, W.: *Multiple-Precision Computations with Result Verification*. In [5], pp 325–356, 1993.

[206] Krämer, W. and Barth, B.: *Computation of Interval Bounds for Weierstrass' Elliptic Function*.

[207] Krückeberg, F.: *Ordinary Differential Equations*. In [127], 1969.

[208] Krückeberg, F.: *Arbitrary Accuracy with Variable Precision Arithmetic*. In [261], pp 95–101, 1985.

[209] Kubota, K., Iri, M.: *PADRE2, a FORTRAN Precompiler Yielding Error Estimates and Second Derivatives*. In [115], pp 251–262, 1991.

[210] Kulisch, U.: *Grundlagen des Numerischen Rechnens — Mathematische Begründung der Rechnerarithmetik*. Reihe Informatik, Band 19, Bibliographisches Institut, Mannheim/Wien/Zürich, 1976.

[211] Kulisch, U., Miranker, W. L.: *Computer Arithmetic in Theory and Practice*. Academic Press, New York, 1981.

[212] Kulisch, U. Ullrich, Ch. (Eds.): *Wissenschaftliches Rechnen und Programmiersprachen*. Berichte des German Chapter of the ACM, Band 10, B. G. Teubner Verlag, Stuttgart, 1982.

[213] Kulisch, U., Miranker, W. L. (Eds.): *A New Approach to Scientific Computation*. Proceedings of Symposium held at IBM Research Center, Yorktown Heights, N.Y., 1982. Academic Press, New York, 1983.

[214] Kulisch, U., Miranker, W. L.: *The Arithmetic of the Digital Computer: A New Approach*. SIAM Review, Vol. **28**, No. 1, 1986.

[215] Kulisch, U. (Ed.): *PASCAL-SC, A Pascal Extension for Scientific Computation*. Information Manual and Floppy Disks. Version ATARI ST. Stuttgart: B.G. Teubner 1987.

[216] Kulisch, U., Stetter, H. J. (Eds.): *Scientific Computation with Automatic Result Verification*. Computing Supplementum **6**. Springer-Verlag, Wien / New York, 1988.

[217] Kulisch, U. (Ed.): *Wissenschaftliches Rechnen mit Ergebnisverifikation — Eine Einführung*. Ausgearbeitet von S. Geörg, R. Hammer und D. Ratz. Akademie Verlag, Berlin, und Vieweg Verlagsgesellschaft, Wiesbaden, 1989.

[218] Kulisch, U.: *Numerik mit automatischer Ergebnisverifikation*. Jahrbuch Überblicke Mathematik 1993, Verlag Vieweg, 1992.

[219] Laifer, R.: *Private communication*.

[220] Lawo, Ch.: *C–XSC — A Programming Environment for Verified Scientific Computing and Numerical Data Processing*. Institute of Applied Mathematics, Prof. Dr. U. Kulisch, University of Karlsruhe, Postfach 6980, D-7500 Karlsruhe, Germany, 1991.

[221] Lawo, Ch.: *C–XSC — A programming environment for eXtended Scientific Computation*. In [326], pp 34, 1991.

[222] Lawo, Ch.: *C–XSC, eine Programmierumgebung für verifiziertes Rechnen in C++*. Talk on SCAN'91 conference, Oldenburg, 1991.

[223] Lawo, Ch.: *C–XSC, eine objektorientierte Programmierumgebung für verifiziertes wissenschaftliches Rechnen*. Dissertation, Universität Karlsruhe, 1992.

[224] Lawo, Ch.: *C–XSC, A Programming Environment for Verified Scientific Computing and Numerical Data Processing*. In [5], pp 71–86, 1993.

[225] L'Ecuyer, P.: *Communications of the ACM*. Vol. 31, pp 742–774, 1988.

[226] Lichter, P.: *Realisierung eines VLSI-Chips für das Gleitkomma-Skalarprodukt der Kulisch-Arithmetik*. Diplomarbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, 1988.

[227] Linnainmaa, S: *Software for Double-Precision Floating-Point Computations*. ACM Trans. on Math. Software, Vol. **7**, No. 3, pp 272–282, 1981.

[228] Lohner, R.: *Enclosing the Solutions of Ordinary Initial and Boundary Value Problems*. In [163], pp 255–286, 1987.

[229] Lohner, R.: *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen*. Dissertation, Universität Karlsruhe, 1988.

[230] Lohner, R.: *Precise Evaluation of Polynomials in Several Variables*. In [216], pp 139–148, 1988.

[231] Lohner, R.: *Enclosing all Eigenvalues of Symmetric Matrices*. In [321], pp 87–103, 1989.

[232] Lohner, R.: *Computation of Guaranteed Enclosures for the Solutions of Ordinary Initial and Boundary Value Problems*. In Cash, J. R.; Gladwell, I. (Eds.): *Computational Ordinary Differential Equations*, pp 425–435, Clarendon Press, Oxford, 1992.

[233] Lohner, R.: *Interval Arithmetic in Staggered Correction Format*. In [5], pp 301–321, 1993.

[234] Lortz, B.: *Eine Langzahlarithmetik mit optimaler einseitiger Rundung*. Dissertation, Universität Karlsruhe, 1971.

[235] Ludyk, G.: *CAE von dynamischen Systemen: Analyse, Simulation, Entwurf von Regelungssystemen*. Springer-Verlag, Berlin, 1990.

[236] *MAPLE, Reference Manual*. Symbolic Computation Group, University of Waterloo (Ontario, Canada), 1988.

[237] Markstein, P. W.: *Computation of Elementary Functions on the IBM RISC System/6000 Processor*. IBM J. Res. Develop., Vol. **34**, No. 1, 1990.

[238] Mayer, G.: *Reguläre Zerlegungen und der Satz von Stein und Rosenberg für Intervallmatrizen*. Habilitationsschrift, Universität Karlsruhe, 1986.

[239] Mayer, G.: *Enclosing the Solutions of Linear Equations by Interval Iterative Processes*. In [216], pp 47–58, 1988.

[240] Mayer, G.: *Grundbegriffe der Intervallrechnung*. In [217], pp 101–117, 1989.

[241] Mayer, G., Frommer, A.: *A Multisplitting Method for Verification and Enclosure on a Parallel Computer*. In [323], pp 483–497, 1990.

[242] Mayer, G.: *Old and New Aspects for the Interval Gaussian Algorithm*. In [164], pp 329–349, 1991.

[243] Mayer, G.: *Some Remarks on Two Interval-Arithmetic Modifications of the Newton Method*. Computing **48**, pp 125–128, 1992.

[244] Mayer, G.: *Enclosures for Eigenvalues and Eigenvectors*. In [26], pp 49–67, 1992.

[245] Meyer, R. K., Schmidt, D. S. (Eds.): *Computer Aided Proofs in Analysis*. Springer-Verlag (The IMA Volumes in Mathematics and Its Applications, Vol. **28**), New York, 1991.

[246] Miranker, W. L., Toupin, R. A.: *Accurate Scientific Computations*. Symposium Bad Neuenahr, Germany, 1985. Lecture Notes in Computer Science, No. **235**, Springer-Verlag, Berlin, 1986.

[247] Moore, R. E.: *Interval Analysis*. Prentice Hall Inc., Englewood Cliffs, N. J., 1966.

[248] Moore, R. E.: *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, Pennsylvania, 1979.

[249] Moore, R. E. (Ed.): *Reliability in Computing: The Role of Interval Methods in Scientific Computing*. Proceedings of the Conference at Columbus, Ohio, September 8–11, 1987; Perspectives in Computing **19**, Academic Press, San Diego, 1988.

[250] Moore, R. E.: *Variable Precision Computing. Strategies for Improving Efficiency*. In [326], pp 73–74, 1991.

[251] Moynihan V.: *Techniques for Generating Accurate Eigensolutions in ADA*. In [321], pp 79–86, 1989.

[252] Mráz, F.: *Nonnegative Solutions of Interval Linear Systems*. In [26], pp 299–308, 1992.

[253] Müller, M.: *Entwicklung eines Chips für auslöschungsfreie Summation von Gleitkommazahlen*. Dissertation, Universität des Saarlandes, Saarbrücken, 1993.

[254] Muller, J.-M.: *Discrete Basis and Computation of Elementary Functions*. IEEE Trans. on Computers, Vol. **C-34**, No. 9, 1985.

[255] Neumaier, A.: *Overestimation in Linear Interval Equations*. SIAM J. Numer. Anal. **24(1)**, pp 207–214, 1987.

[256] Neumaier, A.: *Interval Methods for Systems of Equations*. Cambridge University Press, Cambridge, 1990.

[257] Neumaier, A.: *The Wrapping Effect, Ellipsoid Arithmetic, Stability and Confidence Regions*. In [9], pp 175–190, 1993.

[258] Ng, K. C.: *Argument Reduction for Huge Arguments: Good to the Last Bit*. Sun Pro, Sun Microsystem , Inc_Business, 1992.

[259] Nickel, K. (Ed.): *Interval Mathematics*. Proceedings of the International Symposium, Karlsruhe 1975, Springer-Verlag, Vienna, 1975.

[260] Nickel, K. (Ed.): *Interval Mathematics 1980*. Proceedings of the International Symposium, Freiburg 1980, Academic Press, New York, 1980.

[261] Nickel, K. (Ed.): *Interval Mathematics 1985*. Proceedings of the International Symposium, Freiburg 1985, Springer-Verlag, Vienna, 1986.

[262] Numerik Software GmbH: *PASCAL-XSC User's Guide*. Numerik Software GmbH, Baden-Baden, Germany, 1991.

[263] Payne, M.; Hanek, R.: *Radian Reduction for Trigonometric Functions*. Signum, pp 19–24, Jan. 1983.

[264] Park, S. K., Miller, K. W.: *Communications of the ACM*. Vol. 31, pp 1192–1201, 1988.

[265] Press W. H. et al.: *Numerical Recipes in C, The Art of Scientific Computing*, second edition, Cambridge University Press, 1992.

[266] Rall, L. B.: *Error in Digital Computation*. J. Wiley, New York, 1965.

[267] Rall, L. B.: *Applications of Software for Automatic Differentiation in Numerical Computation*. In [12], pp 141–156, 1980.

[268] Rall, L. B.: *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science, No. **120**, Springer-Verlag, Berlin, 1981.

[269] Rall, L. B.: *Differentiation and Generation of Taylor Coefficients in PASCAL–SC*. In [213], pp 291–309, 1983.

[270] Rall, L. B.: *Differentiation in PASCAL–SC: Type GRADIENT*. ACM TOMS **10**, pp 161–184, 1984.

[271] Rall, L. B.: *Optimal Implementation of Differentiation Arithmetic*. In [163], pp 287–295, 1987.

[272] Rall, L. B.: *Differentiation Arithmetics*. In [322], pp 73–90, 1990.

[273] Rall, L. B.: *Tools for Mathematical Computation*. In Meyer, R. K., Schmidt, D. S.: *Computer Aided Proofs in Analysis*. Springer-Verlag (The IMA Volumes in Mathematics and Its Applications, Vol. **28**), New York, 1991.

[274] Ratschek, H., Rokne, J.: *Computer Methods for the Range of Functions*. Ellis Horwood Limited, Chichester, 1984.

[275] Ratschek, H., Rokne, J.: *Nonuniform Variable Precision Computing*. In [326], pp 71–72, 1991.

[276] Ratz, D.: *The Effects of the Arithmetic of Vector Computers on Basic Numerical Methods*. In [322], pp 499–514, 1990.

[277] Reitwiesner, G.: *An ENIAC Determination of π and e to More Than 2000 Decimal Places*. MTAC, Vol. **4**, pp 11–15, 1950.

[278] Rex, H.-G.: *Zur Lösungseinschließung linearer Gleichungssysteme*. In [151], pp 441–447, 1991.

[279] Rohn, J., Deif, A.: *On the Range of Eigenvalues of an Interval Matrix*. Computing **47**, pp 373–377, 1992.

[280] Rump, S. M.: *Kleine Fehlerschranken bei Matrixproblemen*. Dissertation, Universität Karlsruhe, 1980.

[281] Rump, S. M., Kaucher, E.: *Small Bounds for the Solution of Linear Systems*. In [12], 1980.

[282] Rump, S. M.: *Rechnervorführung, Pakete für Standardprobleme der Numerik*. In [212], pp 29–50, 1982.

[283] Rump, S. M.: *Lösung linearer und nichtlinearer Gleichungssysteme mit maximaler Genauigkeit*. In [212], pp 147–174, 1982.

[284] Rump, S. M.: *How Reliable are Results of Computers? / Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?* In *Jahrbuch Überblicke Mathematik*, Bibliographisches Institut, Mannheim, 1983.

[285] Rump, S. M.: *Solving Algebraic Problems with High Accuracy*. Habilitationsschrift. In [213], pp 51–120, 1983.

[286] Rump, S. M.: *Lineare Probleme*. In [217], pp 119–127, 1989.

[287] Rump, S. M.: *Rigorous Sensitivity Analysis for Systems of Linear and Nonlinear Equations*. Math. Comp. **54(190)**, pp 724–736, 1990.

[288] Rump, S. M.: *Estimation of the Sensitivity of Linear and Nonlinear Algebraic Problems*. Linear Algebra and its Applications **153**, pp 1–34, 1991.

[289] Rump, S. M.: *Convergence Properties of Iterations Using Sets*. In [151], pp 427–431, 1991.

[290] Rump, S. M.: *On the Solution of Interval Linear Systems*. Computing **47**, pp 337–353, 1992.

[291] Rump, S. M.: *Inclusion of the Solution of Large Linear Systems with Positive Definite Symmetric M–Matrix*. In [26], pp 339–347, 1992.

[292] Rump, S. M.: *Validated Solution of Large Linear Systems*. In [9], pp 191–212, 1993.

[293] Ruschitzka, M. (Ed.): *Computer Systems: Performance and Simulation*. In collaboration with Robert Vichnevetsky. Proceedings of 11th IMACS World Congress on System Simulation and Scientific Computation, Aug. 5–9, 1985, Oslo. Preprints see [328]. Elsevier Science Publishers B.V. (North–Holland), 1986.

[294] Salamin, E.: *Computation of π Using Arithmetic-Geometric Mean*. Mathematics of Computation, Vol. **30**, No. 135, July 1976.

[295] Schauer, U., Toupin, R. A.: *Solving Large Sparse Linear Systems with Guaranteed Accuracy*. In [246], pp 142–167, 1986.

[296] Schönhage A., Strassen V.: *Schnelle Multiplikation großer Zahlen*. Computing **7**, pp 281–292, 1971.

[297] Schönhage A., Strassen V.: *Schnelle Multiplikation großer Zahlen*. Computing **7**, pp 281–292, 1971.

[298] Schrage, L.: *ACM Transactions an Mathematical Software*. Vol. 5, pp 132–138, 1979.

[299] Schumacher, G.: *Genauigkeitsfragen bei algebraisch-numerischen Algorithmen auf Skalar- und Vektorrechnern*. Dissertation, Universität Karlsruhe, 1989.

[300] Shanks, D., Wrench, W.: *Calculation of π to 100,000 Decimals*. Math. Computing **16**, 1962.

[301] Shiriaev, D. V. : *On the Memory Efficient Differentiation*. ZAMM **72**, pp 632–634, 1992.

[302] Shiriaev, D. V. : *Reduction of Spatial Complexity in Reverse Automatic Differentiation by Means of Inverted Code*. In [26], pp 475–484 1992.

[303] SIEMENS: *ARITHMOS (BS 2000) Unterprogrammbibliothek für Hochpräzisionsarithmetik. Kurzbeschreibung, Tabellenheft, Benutzerhandbuch*. SIEMENS AG, Bereich Datentechnik, Postfach 83 09 51, D-8000 München 83. Bestellnummer U2900-J-Z87-1, Sept. 1986.

[304] Smith D. M.: *Algorithm 693: A Fortran Package For Floating-Point Multiple-Precision Arithmetic*. ACM Trans. on Math. Software, Vol. **17**, No. 2, pp 273–283, 1991.

[305] Spanier J. and Oldham K. B.: *An Atlas of Functions*. Hemisphere publishing corporation, 1987.

[306] Stetter, H. J.: *Sequential Defect Correction for High-Accuracy Floating-Point Algorithms*. Lecture Notes in Mathematics, Vol. **1006**, pp 186–202, Springer-Verlag, 1984.

[307] Stetter, H. J.: *Staggered Correction Representation, a Feasible Approach to Dynamic Precision*. In Cai, Fosdick, Huang (Eds.): *Proceedings of the Symposium on Scientific Software*, China University of Science and Technology Press, Beijing, China, 1989.

[308] Stewart, G. H.: *Introduction to Matrix Computations*. Academic Press, New York, 1973.

[309] Stoer, J.; Bulirsch, R.: *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.

[310] Storck, U.: *Zweidimensionale Integration mit automatischer Ergebnisverifikation*. ZAMM **73** (7/8), pp T897–T899, 1993.

[311] Storck, U.: *Verified Calculation of the Nodes and Weights for Gaussian Quadrature Formulas*. Interval Computation, St. Petersburg, ISSN 0135-4868, 1993.

[312] Storck, U.: *Numerical Integration in Two Dimensions with Automatic Result Verification*. In [5], pp 187–224, 1993.

[313] Sweeny, D. W.: *On the Computation of Euler's Constant*. Math. Comp. **17**, pp 170–178, 1963.

[314] Tang, P. T. P.: *Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic*. ACM Trans. on Math. Software, Vol. **15**, No. 2, pp 144–157, 1989.

[315] Tang, P. T. P.: *Table-Driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic*. ACM Trans. on Math. Software, Vol. **16**, No. 4, pp 378–400, 1990.

[316] Tang, P. T. P.: *Table-Driven Implementation of the Expm1 Function in IEEE Floating-Point Arithmetic*. ACM Trans. on Math. Software, Vol. **18**, No. 2, pp 211–222, 1992.

[317] Tangelder, R. J. W. T: *The Design of Chip Architectures for Accurate Inner Product Computation*. Dissertation, Technical University Eindhoven, 1992.

[318] Teufel, T.: *Ein optimaler Gleitkommaprozessor*. Dissertation, Universität Karlsruhe, 1984.

[319] Teufel, T., Bohlender, G.: *A Bit-Slice Processor Unit for Optimal Arithmetic*. In [328], Vol. **1**, pp 151–154, [293], pp 325–329, 1986.

[320] Teufel, Th.: *Implementation of a Floating-Point Arithmetic with an Accurate Scalar Product for Digital Signal Processing*. In [26], pp 147–156, 1992.

[321] Ullrich, Ch., Wolff v. Gudenberg, J. (Eds.): *Accurate Numerical Algorithms, A Collection of DIAMOND Research Papers*. Springer-Verlag, Berlin, 1989.

[322] Ullrich, Ch. (Ed.): *Computer Arithmetic and Self-Validating Numerical Methods*. (Proceedings of SCAN-89, invited papers). Academic Press, San Diego, 1990.

[323] Ullrich, Ch.: *Programming Languages for Enclosure Methods*. In [322], pp 115–136, 1990.

[324] Valerio, J.: *Transcendental Approximations Using CORDIC*. 1988.

[325] Varga, R. S.: *Scientific Computation on Mathematical Problems and Conjectures*. SIAM, Philadelphia, 1990.

[326] Vichnevetsky, R., Miller, J. J. H. (Eds.): *IMACS'91, 13th World Congress on Computation and Applied Mathematics. Proceedings in 4 Volumes*. See also [76]. July 22–26, 1991, Trinity College, Dublin, Ireland, 1991.

[327] Volder, J. E.: *The CORDIC Trigonometric Computing Technique*. IRE Transactions on Electronic Computing, Vol. **EC-8**, No. 3, pp 330–334, 1959.

[328] Wahlström, B., Henriksen, R., Sundby, N. P. (Eds.): *Proceedings of 11th IMACS World Congress on System Simulation and Scientific Computation*. Vol. **5**, Oslo, Aug. 5–9 1985. Proceedings published in [293].

[329] Walter, W. V.: *Flexible Precision Control and Dynamic Data Structures for Programming Mathematical and Numerical Algorithms*. Dissertation, Universität Karlsruhe, 1990.

[330] Walther, J. S.: *A Unified Algorithm for Elementary Functions*. Spring Joint Computer Conference Proc., Vol. **38**, 1971.

[331] Weissinger, J.: *Numerische Mathematik auf Personal Computern*. Bibliographisches Institut, Mannheim, 1984.

[332] Weissinger, J.: *Spärlich besetzte Gleichungssysteme. Eine Einführung mit Basic- und Pascal-Programmen*. Bibliographisches Institut, Mannheim, 1990.

[333] Wilkinson, J.: *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

[334] Wilkinson, J.: *Rundungsfehler*. Springer, Berlin, 1969.

[335] Wilkinson, J., Reinsch, C.: *Handbook for Automatic Computation*. Vol. **2**: *Linear Algebra*. Springer Verlag, 1971.

[336] Wolff von Gudenberg, J.: *Einbettung allgemeiner Rechnerarithmetik in PASCAL mittels eines Operatorkonzepts und Implementierung der Standardfunktionen mit optimaler Genauigkeit*. Dissertation, Universität Karlsruhe, 1980.

[337] Wolff von Gudenberg, J.: *Computing z^y with Maximum Accuracy*. Computing **31**, pp 185–189, Springer Verlag, 1983.

[338] Ziv, A.: *Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit*. ACM Trans. on Math. Software, Vol. **17**, No. 3, pp 410–423, 1991.

Index

Index of Special Symbols