ACRITH

High-Accuracy Arithmetic
An Advanced Tool for
Numerical Computation

J. H. Bleher / A. E. Roeder / S. M. Rump

IBM Laboratories

Boeblingen, Germany

ACRITH

Abstract

The High-Accuracy Arithmetic Subroutine Library (ACRITH) is a program product for engineering / scientific application. It consists of a subroutine library for solving problems in numerical computation, such as linear systems, polynomial zeros, eigenvalues, linear optimization etc. All results obtained have algorithmically verified accuracy. An On-line Training Component allows easy familiarization with the capabilities of ACRITH.

Introduction

In the first part of the presentation we will outline the problems engineers and scientists are dealing with. The solution to quite a number of those problems is provided by ACRITH, the High-Accuracy Arithmetic Subroutine Library.

A detailed program description of the ACRITH package will be given in the next part. We will talk about the new instructions which are the base for ACRITH. Performance comparisons with conventional approaches and numerical examples will follow.

The second half of the presentation deals with the discussion and the classification of methods in numerical computation such as symbolic computation, algebraic/infinite computation and interval arithmetic.

The mathematical background with the description of the computer arithmetic published by Kulisch/Miranker [1] and the principles of the new methods [2] used in ACRITH along with examples will complete the presentation.

With existing processors and conventional techniques, it is often difficult, or in some cases not even possible to solve problems in numerical computation to the required degree of accuracy.

Inaccuracies during computation may occur due to:

- loss of digits in small differences through cancellation
- rounding during computation
- conversion of input numbers to machine representation (e.g. from decimal to hexadecimal).

Engineers and scientists envy those users of computers who are sure that in their programs no such problems can occur.

Another category of users are those who have recognized, that they have numerical problems and try to overcome those. They have quite a hard job to do with the analysis of their input values, with the selection of specific algorithms that are designed for this special application. The

error analysis requires a high mathematical skill and ends sometimes in costly reruns with altered input parameters or higher precision.

The last part of the user community is even not aware that its results might be wrong. Those users may have heard once in a while that in floating-point computation inaccuracies can occur, but they believe that those are very rare cases. Very often their prime objective is the performance they can get and they don't care about the accuracy. They want a fast result knowing that it might be wrong.

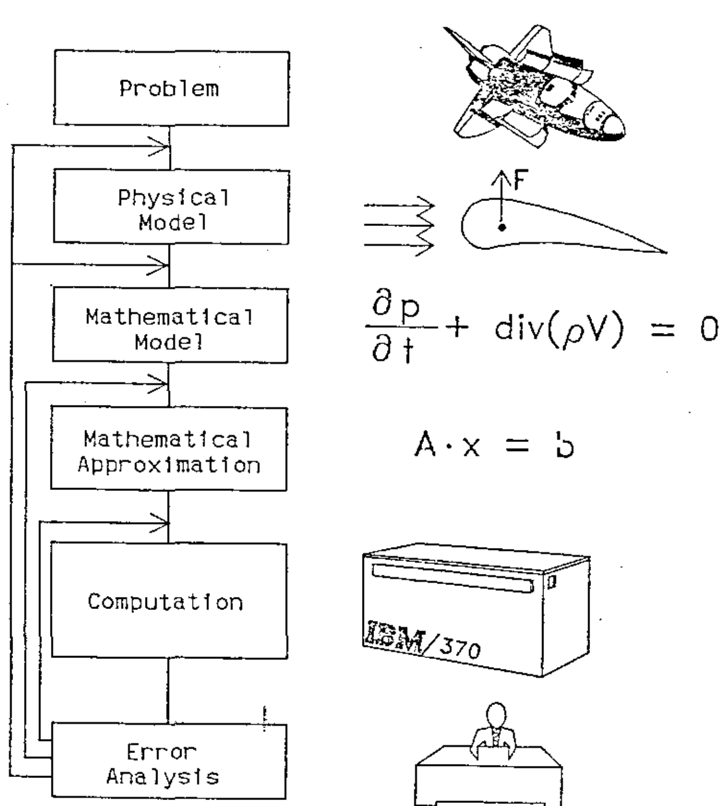

Figure 1: Solution of real-world problems, conventional

Figure 1 shows the procedure used in the engineering/scientific environment when solving real-world problems.

The first task is the description of the problem. Although most engineers and scientists immediately jump into the physical or mathematical world the clear description of the problem with all the boundary conditions is a key element for a successful continuation of the work.

The second step is the description of the physical model. The example in Fig. 1 shows the profile of a wing which has to be investigated. Knowledge and experience leads from the physical level to the description of the problem by mathematical equations. Unfortunately those equations normally are not adequate for the computational phase. Their complexity again requires hard brain work to get equations that can be handled by

standard routines available on our computers. We call this part the mathematical approximation.

With all those preparations we are now ready to feed the computer.

In our example the mathematical approximation results in a linear equation described by the equation A . x = b where A is a matrix, x the vector of the unknowns and the vector b is called the right hand side.

With the results on the screen or with a printout of the results brain work starts again. A difficult and often a time consuming task is a detailed error analysis. If the result is not correct, different "loops" for a correction are possible. Along the inner loop the computation is executed again with a change in the precision of the floating-point computation for example from short to long or even to extended precision. Thus the number of digits used during computation is increased and there is a higher probability for a better result. If, after a repeated error analysis the result is not satisfactory, the professional has a different algorithm ready to deal with this "ill-conditioned problem". He is changing his program using the new algorithm and runs the problem again.

Now, let's assume that this time he concludes his error analysis with the strong feeling that he can trust his results. Otherwise the two outer loops would have asked him to think over his physical or mathematical model and start all the work again.

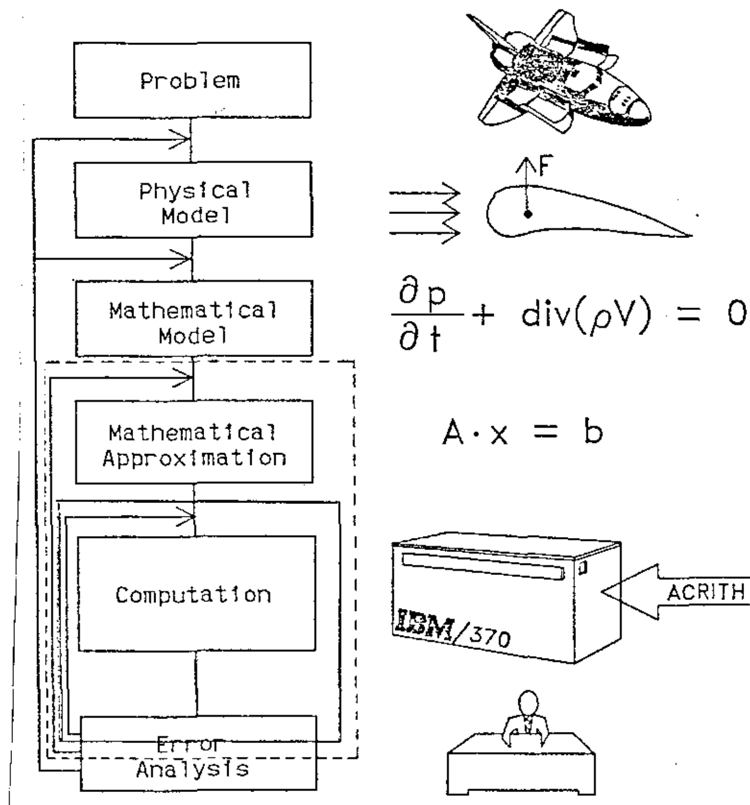

Figure 2: Solution of real-world problems, with ACRITH

Figure 2 shows where ACRITH can help the user to complete his task. ACRITH directly influences the computational phase. For the given input parameters the solution is determined with verified accuracy, in this example the solution of the system of linear equations.
Therewith the numerical part of the error analysis is no longer necessary.

In addition to that the knowledge of the capabilities of ACRITH influences the mathematical approximation. This is depicted by the dashed lines in Fig. 2. Some approximation methods which deliver good results have not been used in the past due to problems during numerical computation. Having in mind that ACRITH delivers results of verified accuracy some approximation methods can be successfully used again.

ACRITH Description

The High-Accuracy Arithmetic Subroutine Library is an IBM-First. It consists of routines for solving problems in numerical computation. All results obtained have an algorithmically verification of the correctness and the accuracy.

If the problem is extremely ill-conditioned, this means that for example the matrix of a linear system is singular or very close to singularity the user is informed of this fact by an appropriate return code.

The capability of ACRITH of dealing even with extremely ill-conditioned problems is, for instance, demonstrated by inverting a Hilbert 21 by 21 matrix. This is, after multiplying with a proper factor, the largest Hilbert matrix exactly storable in /370 long format.

Together with the ACRITH Subroutine Library a so-called On-line Training Component (OTC) is provided. The OTC has been designed to give the user a valuable tool for familiarization with the capabilities of ACRITH. In addition to that it allows solving of numerical problems interactively.

ACRITH runs on all System /370 processors under VM/SP. It is callable from VS Fortran and Assembler programs. ACRITH arithmetic, which bases on a sound theory by Kulisch and Miranker makes use of the High-Accuracy Arithmetic Facility. This architecture RPQ provides 20 new instructions, with rounding. The microcode implementation on all 4361 processors results in a remarkable performance improvement of the arithmetic and the subroutines when running on one of those processors.
With the ACRITH package a software simulation written in Assembler for the new instructions is provided enabling the subroutines to run on all System /370 processors.

Subroutine Library

The ACRITH subroutine library offers a large variety of routines for solving problems in numerical computation. The highest level of the routines are called the problem solving routines. They deal with:

- arithmetic expressions
- polynomial evaluation
- zeros of polynomials
- linear equations
- matrix inversion
- linear programming
- eigenvalues and eigenvectors.

The routines in the next lower level are the basic arithmetic routines:

- conversion (decimal <---> hexadecimal)
- vector operations
- scalar product
- matrix multiplication

The lowest level routines provide access to the architected long accumulator which will be described later in this presentation. These routines allow the following operations:

- add scalar product to accumulator
- add/subtract accumulator
- store accumulator
- clear accumulator

Environment

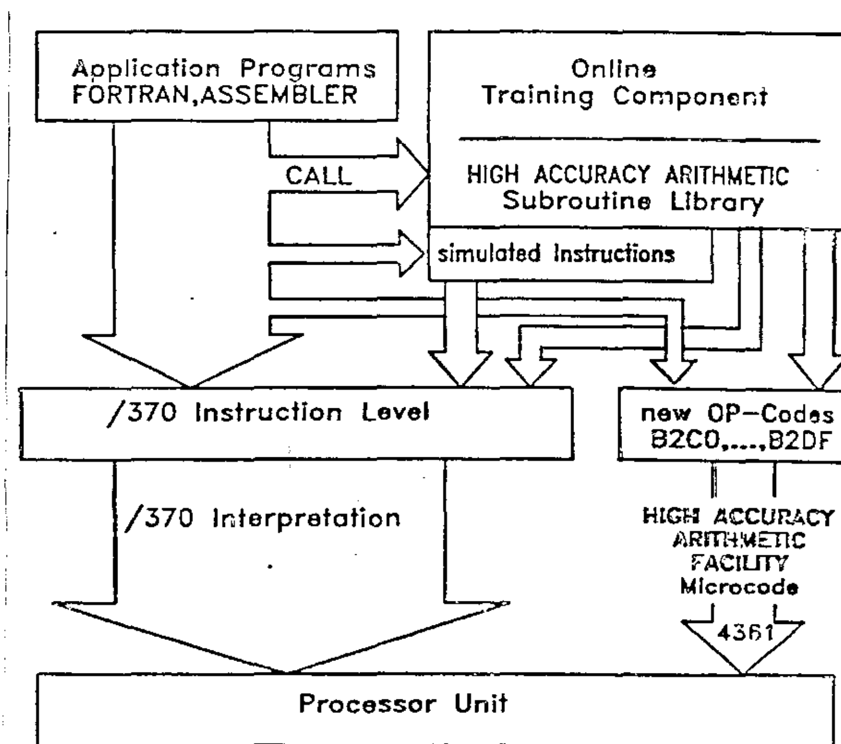

Figure 3: ACRITH application environment

The ACRITH application environment is depicted in Figure 3. In the upper left corner the application programs written in FORTRAN and/or Assembler are shown. After compilation /370 instructions are executed to obtain results.
To take advantage of the capabilities of ACRITH, parts of the old programs respectively calls to other libraries have to be substituted by a "CALL" to the ACRITH subroutines. The subroutines execute on standard /370 level and use in addition the 20 new instructions, either directly on /370 machine level on all 4361 processors, or, via the simulated instructions on the standard /370 level.

Advanced users may directly use the 20 instructions to write their own programs. This possibility is also shown in Figure 3 by the small horizontal arrows.

At this point it should be mentioned that an application program using ACRITH runs on all /370 processors. If the microcoded instructions of the ACRITH Facility are available (4361 processor), the subroutines automatically use them, without the necessity of recompilation or relink, i.e. ACRITH programs always choose the fastest mode.

ACRITH Facility

The new and outstanding capabilities provided with the ACRITH package base on the architectural definition of 20 instructions.

These instructions can be divided in two classes, namely the basic arithmetic instructions and the accumulator instructions.

The basic arithmetic instructions consist of:

- add with rounding
- subtract with rounding
- multiply with rounding
- divide with rounding
- load with rounding

Four possible roundings are available:

- rounding upwards (towards + infinity)
- rounding downwards (towards - infinity)
- rounding to nearest floating-point number
- rounding to zero

All instructions are defined and architected for System /370 short and long floating-point format. The results obtained with these instructions are all of maximum accuracy, which means that, within the given floating-point format used, no floating-point number lies between the computed result and the result obtained with infinite precision. A key point of the Kulisch/Miranker theory is the introduction of an additional instruction: the scalar product with maximum accuracy. This is the step from the maximum accuracy single operation to maximum accuracy composed operations.

The instructions defined in conjunction with the scalar product are called accumulator instructions:

- add/subtract operand to/from accumulator
- multiply and accumulate (scalar product)
- round from accumulator
- add/subtract accumulator to/from accumulator
- clear accumulator

The long accumulator occupies a 168 byte storage area. The layout of an accumulator in storage is depicted in Figure 4.

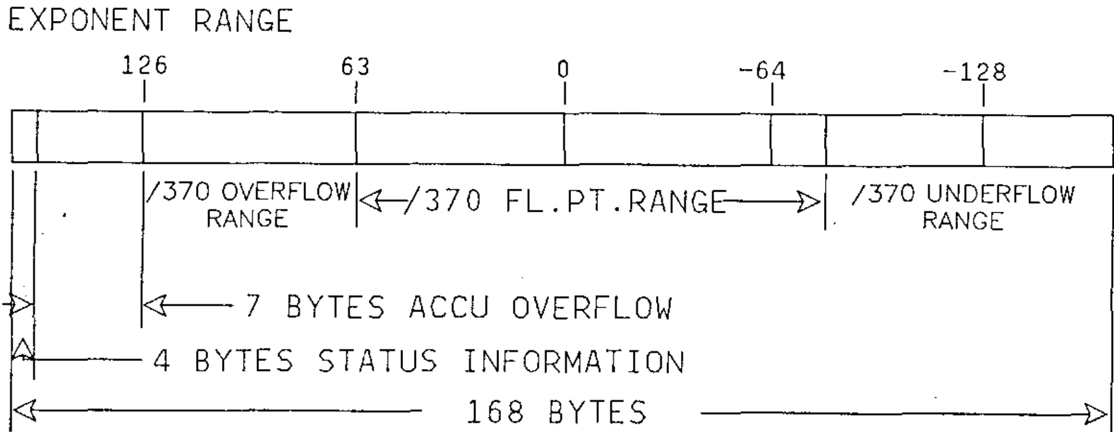

Figure 4: ACRITH accumulator layout

The accumulator consists of a four byte status area on the left, followed by a 164 byte numeric area. When a floating-point number is added to the accumulator, the fraction is positioned in the numeric area that corresponds to the exponent. The fraction is added at that point, and any carries are propagated to the left as far as necessary.

The exponent range covered in the accumulator is 16 ** 126 through 16 ** (-128), which is twice the standard /370 exponent range. Together with 14 digits for accumulator overflow no exponent overflow can occur, because the numeric area is large enough to allow any reasonable number of scalar products of the largest representable floating-point numbers to be accumulated.

Performance

Three levels of complexity are distinguished when comparing program execution times. The lowest level is the machine level with standard /370 instructions and the 20 new instructions defined in the High-Accuracy Arithmetic Facility.

For comparison we execute a scalar product with a conventional assembler loop, i.e. without any check or verification of accuracy. The same scalar product then is executed with the new instruction "multiply and accumulate", which delivers in any case results of maximum accuracy. If the ACRITH instruction is implemented in microcode (4361 processor) approximately the same execution time is encountered. In case of software simulation of the ACRITH instruction a performance degradation of a factor of 2 has to be paid for the advantage of maximum accuracy.

Next, performance comparison is done on the procedure level. When executing ACRITH routines (e.g. linear system solver) in long

floating-point format on a 4361 processor a similar run time is measured as in case of conventional routines (without verified accuracy) in extended precision. If the software simulation is used (on all other /370 processors) the run time is approximately twice as long with ACRITH.

Performance comparison on the application level is not just a figure or a ratio. The comparison of ACRITH solutions versus conventional solutions clearly shows the advantage of ACRITH in terms of saving human and machine time. As already discussed in the introduction when going through the "real-world problem", ACRITH removes a significant burden from the user. The following steps are drastically reduced by the introduction of ACRITH, or in some cases they are totally superfluous:
- input data analysis - time consuming error analysis - change of algorithm (mathematical approximation) - costly reruns.

Application Examples

The following list gives a brief overview of potential application areas of ACRITH.

- construction - structural analysis - fluid mechanics - statistics - circuit design - energy conversion - power distribution - nuclear fusion - robotics - aviation - space navigation

Most of the applications have been mentioned to us during our presentations or in discussions with engineers and scientists who deal with numerical computation.

Examples

In order to demonstrate the capabilities of ACRITH we have selected two examples. The first problem deals with a set of linear equations. The solution of the linear system.

$$37639840\ X - 46099201\ Y = 0$$
$$29180479\ X - 35738642\ Y = -1$$

is obtained with ACRITH as

$$X = 46099201 \text{ and } Y = 37639840.$$

This result is verified to be correct as well as the matrix is automatically verified to be non-singular.
With a conventional approach, using the 'Gaussian' elimination method one would have obtained three different results depending on the precision:

single:	no result (divide exception)
double:	X = 41095618.5 Y = 33554432
extended:	X = 46099201 Y = 37639840.

In case of single precision the divide exception alerts the user that something is wrong with his problem. The results obtained with double and extended precision differ already in the second digit. It is up to the user, to verify by time consuming error analysis that the extended result is correct.

ACRITH, however delivers the result including the verification step, which means that the user can trust his result.

The second example deals with the evaluation of an arithmetic expression.

$$83521X^8 + 578X^4 Y^2 - 2Y^4 + 2Y^6 - Y^8$$

The evaluation of this expression with ACRITH at

$$X = 2289912 \text{ and } Y = 9478657$$

delivers the verified result of

$$- 1.79689877047297 E + 14$$

A conventional program would result in

single:	- 2.33840262....	E + 50
double:	- 5.445178707...	E + 39
extended:	- 1.511157274...	E + 23

All results differ from the exact result in orders of magnitude, since the exponents of the results are different. The result in short precision is wrong by a factor of 10^{36} whereas the result in extended precision is still wrong by a factor of 10^9, that is wrong by 100000000000 %.

Historical Background

In the historical background we want to give an overview on methods used in the past to obtain accurate or exact results. The methods have been derived after users realized the problems with roundoff errors and cancellation and the resulting errors in numerical computation.
The different approaches are:

- Symbolic computation
- Algebraic computation
- Naive interval arithmetic

The three methods were invented in the early sixties. The first two are strongly connected and they are often together referred to as "Symbolic and Algebraic Manipulation." The research and development in this area is in steady progress.
The third method, the naive interval arithmetic, was proposed as a global solution to numerical problems. It turned out that this is not the case. The naive interval arithmetic has been finished after few years whereas the sophisticated interval mathematics has been settled as an individual part of numerical analysis.

The standard floating-point algorithms do not deliver verified results (as has been demonstrated by examples) or provide error bounds for the results. The delivered results are often of high accuracy, but sometimes vastly wrong.

The new methods ACRITH is based on, deliver always results which are verified to be correct, i.e. no wrong results are possible. Moreover, the results of the lower level algorithms are always of maximum accuracy. The results of the higher level algorithms are almost always of maximum accuracy. The property of maximum accuracy means, that between the computed result and the infinite precise result there is no other floating-point number.

Symbolic Computation

The principle of symbolic computation is, that symbols are manipulated as in mathematical hand calculation. For example, the calculation of

$$(x+2) * (x-5)$$

is executed as human beings perform it on paper following the rules of algebra. In a symbolic computation (using list processing facilities to store the symbols adequately) the result would be

$$x*x + 2*x - 5*x - 10$$

The second phase of a symbolic computation is the simplification. This task is rather involved. In the past years significant improvements have been made to perform simplification strategies resulting in a new mathematical direction called universal algebra. A final step is the two-dimensional output to show an expression as in mathematical hand calculation, in our example

$$x^2-3x-10 \ .$$

If the whole calculation is performed symbolically without substituting variables by actual values, the complexity of expressions tend to grow very rapidly.

In our example the inverse of m*m is computed, where m is the 2x2 matrix with entries a, b, c, d. The square m*m of the matrix m is fairly easy to compute, the result is

$$m*m = \begin{matrix} * & a^2 + bc & ab + bd & * \\ * & & & * \\ * & ac + cd & bc + d^2 & * \end{matrix}$$

This could be done by hand calculation. The inversion of m*m is an involved process and at least for matrices with 3 rows and more the inverse can hardly be computed by hand. The computer does it automatically without the error probability when computing by hand.

Symbolic computation has gained in interest by new packages for symbolic integration. These algorithms are able to verify on a sound mathematical background, whether a transcendental expression is integrable or not. This and the other features of symbolic computation have become important tools for the mathematician.

However, it requires a lot of computing time to handle symbolic expressions. Because the expressions tend to grow rapidly in size symbolic manipulation is hardly applicable for solid numerical problems. So the next step is to handle expressions not symbolically but algebraically, i.e. first inserting values for the variables and then computing exactly, that means in infinite precision.

Algebraic Computation

The key feature of algebraic computation is, that each arithmetic operation is performed exactly, that is without error. The process of performing the individual operations addition, subtraction, multiplication, division etc. is the same as in hand calculation according to the rules of algebra. For example, in case of multiplication one operand is multiplied by every digit of the other operand and the intermediate products are added. The resulting product may consist of many digits which have to be handled properly. For example, multiplying a 10-digit number by itself results in a 20-digit number, multiplying this again by itself results in a 40-digit number and so forth. Therefore algebraic computation requires an adequate memory organization for handling many digits of a number.

In our example we performed 200! / 2 ** 500. This is quite a task. The numerator, 200! (read 200 factorial, that is 1*2*3*4*...*199*200) is a number with 375 decimal digits. The denominator 2**500 is a number with 151 decimal digits. When dividing both, the greatest common divisor of numerator and denominator is computed, that is of a 375-digit and a 151-digit number. The result is a fraction with a 315-digit numerator and a 92-digit denominator.

Algebraic computation deals not only with integers or rational numbers but also with algebraic numbers, transcendental numbers etc. For instance, it is possible to store the square root of 2 on the computer without rounding error. This square root of 2 is the zero of the polyno-

mial x^2-2 between 1 and 2, and in this description only integer numbers are involved. It is possible to perform calculations with these algebraic numbers without rounding errors and without cancellation errors.

Algebraic computation is in fact an infinite precision arithmetic. Whenever the memory of the machine suffices to store the result of an operation this result is mathematically correct without error, it is "infinitely precise". In a computation the number of digits of intermediate results tend to grow very rapidly. When multiplying a number by itself the number of figures of the result doubles.

In our example we invert a 4 x 4 matrix where the entries are 2-digit numbers, whereas the result consists of 13-digit rational numbers.

Example: Inversion of 4x4 matrix

89	-23	-31	47
-33	59	-7	-43
53	32	25	-61
83	-97	31	17

Solution in infinite arithmetic:

7983/1346420	-2468/336605	1802/201963	-11803/4039260
-671/336605	-15446/336605	4513/201963	-30674/1009815
-6696/336605	-58957/673210	7312/201963	-73933/2019630
-5449/1346420	-44657/673210	3619/201963	-135221/4039260

Therefore algebraic computation is not suitable for numerical computations. Even very small numerical computations performing only some thousand operations are hardly executable in algebraic computation. Beside the fact, that the memory of the computer would not suffice to store the usually very long intermediate results, the computing time is much higher than that of a comparable floating-print algorithm (the latter, of course, cannot provide exact results).

Naive Interval Arithmetic

The first principle of naive interval arithmetic is that numbers or variables are stored as entities with a certain tolerance. This can be done in an engineering notation

$$3.14159 \pm 0.000005,$$

or in a notation giving a minimum and a maximum value

$$[\ 3.14159\ ,\ 3.14160\ ]\quad .$$

The latter notation expresses that the number p we have in mind is somewhere between 3.14159 and 3.14160, or mathematically

$$3.14159 \leq p \leq 3.14160.$$

These bounds for a certain number are called intervals. Given two intervals, it is possible to perform computations with them regarding the tolerances. Suppose we have to perform

$$[\ 1.1\ ,\ 1.2\ ] + [\ 0.2\ ,\ 0.3\ ]\quad .$$

Then the naive interval addition is a worst case analysis: What are the bounds for the result when taking any (real) number out of the first interval and adding any (real) number out of the second interval? The smallest possible result is adding 1.1 and 0.2, the smallest values in of either intervals. The largest possible result occurs when adding the largest values of the two intervals, thus the result is

$$[\ 1.3\ ,\ 1.5\ ]$$

It can be shown that all intermediate results are possible, that is for any (real) number c between 1.3 and 1.5 two numbers out of the first and second summand interval can be found the sum of which is c.
It is possible to perform this worst case analysis for the four basic operations: addition, subtraction, multiplication and division. However, the left and right bound of the result of an operation is not always the operation of the left and right bounds as in the example above.
Consider

$$[\ -0.1\ ,\ 0.2\ ] * [\ 4.7\ ,\ 4.8\ ]$$

The left bound of the result

$$[\ -0.48\ ,\ 0.96\ ]$$

is multiplying the left bound of the first with the right bound of second interval, the right bound is multiplying the right bound of the first again with the right bound of the second interval. So, in this case, the left bound of the second interval was not involved in the computation.

The worst case analysis of naive interval arithmetic leads to difficulties when the same number occurs more than once in a computation.

Consider the following example:

$a = 1/3$
$b = 1/3$
Compute the value of $a - b$.

It is clear that the result of b - a is zero. But what happens in naive interval arithmetic? Suppose we perform the computation on a 5-digit computer. Then the closest result for 1/3 is [0.33333, 0.33334] due to the limited number of figures in the computer. It should be pointed out that on any computer, no matter how many digits it has, the principle of the example would be the same. Therefore, we have on the 5-digit computer

the value for a [0.33333, 0.33334].

and

the value for b [0.33333, 0.33334].

However, with these two interval results the connection to the original computation 1/3 is totally lost. In fact, the values for a and b might have come from different computations and have therefore to be treated independently. And that is the key point: Dependencies in the computations cannot be recognized. Therefore, when subtracting a and b, we have to consider the worst case. And that is, that the true value for the variable a might be the left bound 0.33333 and the true value for b might be the right bound 0.33334. This has to be done to remain on the safe side, because the additional information that a and b come from the same source (and are in fact identical) is lost.
Thus the result is

$$a - b = [- 0.00001, + 0.00001].$$

These overestimations occur frequently in numerical computation. When, for example, inverting a 10 x 10 matrix, 95% out of the approximately 2000 operations performed are such, were dependencies are involved. Therefore, the width of intermediate result intervals tend to grow very rapidly such that the final result is often of not very much value.

But there is still another effect. When in a computation a division has to be performed by an interval and the width has grown that far, that this interval contains 0, then the computation has to be stopped because division by 0 is not possible. Thus we have three cases of results of naive interval arithmetic compared with standard floating-point arithmetic:

naive interval arithmetic	standard floating-point
1) sharp bounds	good approximation
2) wide bounds	may be good, may be bad
3) no result (division by zero)	may be good, may be bad

In those cases, where a better method would be of real need we either get wide bounds or no result using naive interval arithmetic.

Floating-point number system

In the first digital computer a fixed point arithmetic was implemented. As in analog computers only numbers of magnitude less than one were allowed and every problem had to be scaled such that it fit in that number range. When floating-point arithmetic was introduced many people warned
that due to a large exponent range rounding and cancellation errors may falsify a computational result significantly. The classical paper of Goldstine and Neumann [3] points in that direction.

In a floating-point number system on todays computers the mantissa is represented in binary or hexadecimal format. There is a first difficulty when using floating-point numbers, that is the conversion between decimal and binary format (with respect to the principle problem of conversion or accuracy there is no difference between binary and hexadecimal format). Many decimal numbers are not exactly convertible in binary format no matter how many digits are used in the mantissa. A simple example is the decimal number 1.6. The representation in binary format is

1.10011001100110011 ....

continuing for ever. On a computer this infinite sequence of digits has to be chopped somewhere introducing a small error. These errors are usually small, but they might accumulate in a computation up to a catastrophic error for a final result.

Computer Arithmetic, traditional

The floating-point arithmetic on digital computers is designed to approximate the true (infinite precise) result as good as possible. Spoken in decimal, the result of 2/3 should be 0.6666...667 in a certain accuracy because this is the nearest finite decimal to 2/3.

The implementation of the floating-point arithmetic is due to the computer manufacturer. From there usually no information is available on the accuracy of the arithmetic operations addition, subtraction, multiplication and division. Usually, the computed results approximate the

infinite precise result fairly good, but the results on different machines are not the same even if the numbers have the same floating-point format.

Our example refers to System /370 short format, but similar examples can be found on any machine. As pointed out the number 1.6 is not exactly representable in binary (and hexadecimal) format. That means, that writing

A = 1.6

in a program does not imply that the value 1.6 is stored in the variable A but a slightly different (rounded) value. In our case the last bit was rounded upwards because the following bit is a 1 (that is like rounding .5 upwards in decimal). The next assignment

B = 8.0 / 5.0

gives the same infinite precise result, namely 1.6 in decimal. On the computer, again, the result has to be rounded because it is not exactly representable. In this case the result is truncated yielding a difference between A and B in the last bit. Subtracting both yields a difference of $5 \cdot 10^{-7}$. The difference is fairly small but may accumulate to large errors.

The critical point is, that the implementation of the floating-point arithmetic and the conversion between decimal and hexadecimal and vice versa is due to the computer manufacturers. The latter, the conversion, is computer dependent and results may even differ between different releases of the same compiler on the same machine.

ACRITH Computer Arithmetic

The first principle of the ACRITH computer arithmetic is, that any single operation is performed with maximum accuracy. This holds for the basic arithmetic operations addition, subtraction, multiplication and division and for conversion from decimal to hexadecimal and vice versa.

We distinguish different rounding modes: the rounding to nearest, downwards, upwards and towards zero. The result rounded downwards, for instance, is the largest floating-point number being less than or equal the infinite precise result. In our example of a 5-digit decimal computer, the result of 2/3 rounded downwards is 0.66666. The ACRITH computer arithmetic performs the mentioned operations with maximum accuracy under any circumstances. For example, the result of the conversion from decimal to haxadecimal of

0.000000000000000000000847032947254300339068322500679641962051391601 5625

will be 2^{-70} in any rounding mode, that is the decimal number is exactly converted to hexadecimal format. (The example should not encourage the user to compute with numbers of that size, it should demonstrate the

fact, that maximum accuracy is achieved in ACRITH under any, even extreme circumstances).

The maximum accuracy property for the four basic arithmetic operations and the conversion guarantees a result of maximum accuracy for any single operation or assignment. However, this need not be true for several operations. Consider for example the following scalar product:

```
2.718281828  *  1486.2497    -
3.141592654  *  878366.9879  -
1.414213562  *  22.37492     +
0.5772156649 *  4773714.647  +
0.3010299957 *  0.000185049
```

The correct value for the scalar product is $-1.0065 \cdot 10^{-11}$. The reader is encouraged to try this example on a pocket calculator and/or a large computer. The computed approximation will almost always be vastly incorrect if the accuracy of the computer is not at least 17 digits.

It turns out, that highly or even maximum accurate single operations do not suffice to perform accurate computations. A closer look at the spaces in numerical computation gives a variety of higher spaces such as

- vectors
- matrices
- complex numbers
- complex vectors
- complex matrices
- intervals
- interval vectors
- interval matrices
- complex intervals
- complex interval vectors
- complex interval matrices

Beside the basic operations within these spaces there are many "outer" operations, i.e. operations between elements of different spaces. Counting the number of all these operations yields a surprising number of over 600.

It has been shown by the theory of Kulisch and Miranker [1], that all these operations can be performed with maximum accuracy when a scalar product for real numbers with maximum accuracy is available. Because of the outstanding importance of the precise scalar product we refer in the ACRITH arithmetic to

5 basic arithmetic operations, namely
addition, subtraction, multiplication, division, scalar product.

A maximum accurate computer arithmetic for these 5 basic arithmetic operations is a sound basis for computation in higher spaces in numerical mathematics. The precise scalar product represents the step from the accurate single operation between numbers to accurate (single) operations in higher numerical spaces. The essential progress of ACRITH is the step from the single operation in higher numerical spaces to problems in numerical computation, to real world problems. The sound mathematical basis for solving problems in numerical computation is the inclusion theory [2].

New Methods

Let's consider the case of a system of linear equations. In standard numerical methods a verification of results is not possible. To improve the quality of an approximation a "residual iteration" is applied. In theory this is a steady improvement of a given approximation, the limit of which is the solution of the linear system.

In practice, the computation is not performed precisely (in infinite precision), but in floating-point arithmetic. The introduced rounding errors may falsify the intermediate steps of the iteration so that it finally does not converge. On the other hand, a convergence in a mathematical sense (computed in infinite precision) implies, that the given problem is solvable and that the iteration point is the solution. This is not the case for a numerical convergence using floating-point arithmetic.

Therefore a key problem is the development of a criterion to decide, whether an iteration converges or not. The next key question related to that is, whether a solution exists and whether it is unique. This problem is mathematically equivalent to the question, whether the matrix of the linear system is not singular. This has, up to know, hardly been solvable using floating-point arithmetic.

A respecting criterion is established by the new inclusion theory [2]. This theory contains theorems the assertions of which are exactly what we need:

- The iteration converges
- There exists a solution
- The solution is unique
- An inclusion of the solution is provided.

This applies to linear as well as to nonlinear problems. We continue referring to the linear case. The theorems use a new form of the residual iteration:

$$X^{k+1} = R*b + (I - R*A) * X^{k} ,$$

where A is the matrix of the linear system and b is the right hand side. I is the identity matrix and R an approximate inverse of the matrix A. R may be represented as a triangular decomposition. Note, that there is no assumption on the non-singularity of the matrix A or the matrix R. This new residual iteration applies to sets of vectors rather than to single vectors. An inclusion of one iterate in the next guarantees the assertions stated above.

The verification of the inclusion of a set in another set and the calculation of the new residual iteration, the calculation with sets on computers requires a new arithmetic. That is the correspondence between the new inclusion theory and the new arithmetic. The arithmetic is used to verify the assumptions of the theorems of the inclusion theory. Then the assertions of the theorems are valid such as existence, uniqueness and inclusion of the solution. ACRITH represents the implementation of this combination between the new arithmetic and the new inclusion theory.

There is still a lot of work to do to obtain not only an inclusion of the solution but sharp inclusions, which are mostly of maximum accuracy. These implementation details represent the step from the inclusion theory to the practical algorithm, to the subroutine library.

The key feature of the new algorithms is that every result is automatically verified to be correct. Another novel feature is that it is possible to solve problems where the data is afflicted with tolerances. If some input data is the output, for example, of a meter, then only a limited number of digits is correct, the rest is in the tolerance level. In standard floating-point arithmetic there is no way to handle inaccurate data. The only chance is to pick some sample values out of the tolerance interval, for instance using Monte Carlo methods. These methods cannot give an overview over all possible solutions, they cannot show whether the problem is solvable for all possible combinations of input data and the methods are extremely time consuming.

A simple example can demonstrate this fact. Suppose, a 10 x 10 matrix has to be inverted. If only 10 % of the 100 entries is afflicted with a tolerance, and if only the extreme value of each tolerance are computed, then there are still 2^{10}, that is over 1000 different possibilities. The computing time only for the extreme cases is 1000 times solving the problem. And still only the extreme values are covered and the infinitely many possibilities within the tolerances are not taken into account.

With ACRITH it is possible to solve problems where the data is afflicted with tolerances. Virtually, all infinitely many problems within the tolerances are solved. In practice, an inclusion is computed containing all solutions of all possible problems within the tolerances. If within the tolerances, within these infinitely many problems, there is only one problem which is not solvable, this fact is reported to the user.

ACRITH - examples

We first show a problem of data afflicted with tolerances.

Example: Given a 3x3 matrix, coefficients with tolerances

```
      5.24754155          (-2.15251107,-2.15251104)     -0.52157255
 (8.62972789,8.62972795)       -5.84425364         (1.76003657,1.76003663)
(13.01191425,13.01191434)      -9.53599622                 4.0416457
```

Question: Is the matrix invertible for any combination of tolerances?

Answer by ACRITH:

Yes, it has been automatically verified, that all matrices within the tolerances are invertible and all inverses are included in:

```
row  1 column  1:  (   0.99999984D+00 ,  0.100000022D+01 )
       column  2:  (  -0.20000006D+01 ,  -0.19999996D+01 )
       column  3:  (   0.99999983D+00 ,  0.100000032D+01 )

row  2 column  1:  (   0.17518473D+01 ,   0.17518481D+01 )
       column  2:  (   -0.4094866D+01 ,   -0.4094863D+01 )
       column  3:  (   0.20092867D+01 ,   0.20092878D+01 )

row  3 column  1:  (    0.9139092D+00 ,    0.9139099D+00 )
       column  2:  (   -0.3222645D+01 ,   -0.3222642D+01 )
       column  3:  (   0.17687446D+01 ,   0.17687454D+01 )
```

The inversion with ACRITH gives an inclusion of all matrices included within the tolerances. Due to the fact that the input data is only correct to 10 figures, the inclusion cannot be more accurate.

In the second example only the first digit of the very first component of the matrix is changed from 5 to 4.

Example: Given a 3x3 matrix, coefficients with tolerances

```
      4.24754155          (-2.15251107,-2.15251104)     -0.52157255
 (8.62972789,8.62972795)       -5.84425364         (1.76003657,1.76003663)
(13.01191425,13.01191434)      -9.53599622                 4.0416457
```

Question: Is the matrix invertible for any combination of toloerances?

Answer by ACRITH:

No inclusion computed. The interval matrix probably contains a singular matrix.

Now, somewhere within the tolerances, there is a singular matrix which is not invertible. The user is informed of this fact which would be impossible to find out trying some combinations of tolerances.

The last example poses the question whether the polynomial

$$p(x) = 8118.0\,x^4 - 11482.0\,x^3 + x^2 + 5741.0\,x - 2030.0$$

has a zero near 0.7 or not. The answer with ACRITH is that between

0.707070707070707070

and

0.707070707070707071

there is a zero of the polynomial. This is computed in System /370 long precision and is a result of maximum accuracy. In figure 5 we show a small graph around the zero where the values indicated by an asterisk are computed by ACRITH.

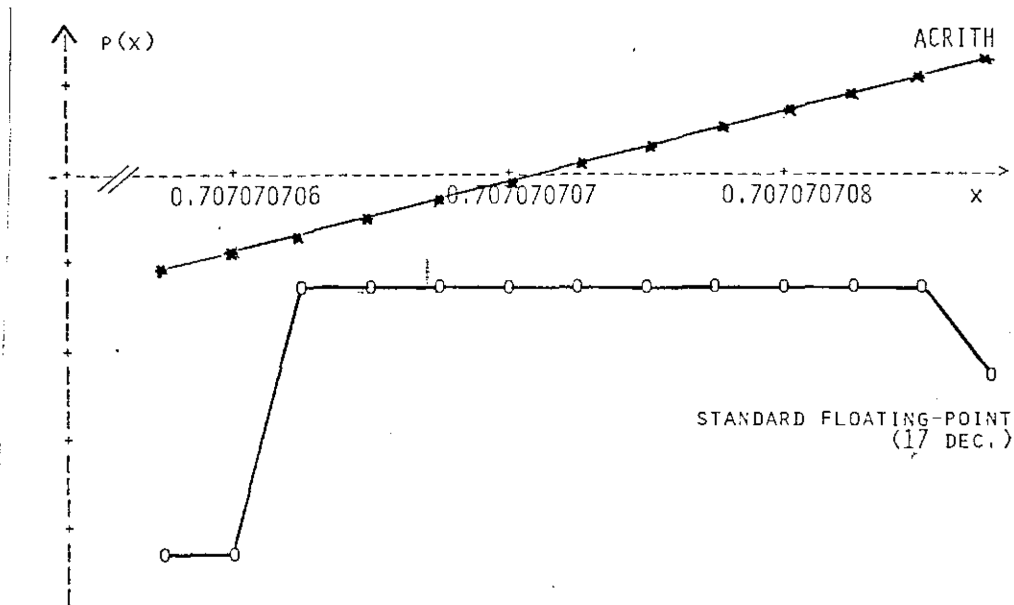

Figure 5: Graph of p(x)

Using a standard floating-point algorithm (Horner's scheme), we obtain in System /370 long precision the dotted results. This result indicates that there is no zero of the polynomial near 0.7.

However, when computing zeros of polynomials, these regions where the value of the polynomial is small, have to be examined. These are some of

the regions where cancellations may alter an approximation of the value of the polynomial significantly up to totally false results.

Summary

With the High-Accuracy Arithmetic Subroutine Library (ACRITH) a new dimension in numerical computations is entered. The algorithmically verified results delivered by the subroutines remove a significant burden from the user.

References

[1] Kulisch, U., Miranker, W. L.: Computer Arithmetic in Theory and Practice. Academic Press, New York (1981).

[2] Rump, S.M.: Solving algebraic Problems with High Accuracy, 76 pages, in "A New Approach to Scientific Computation", August 1982. Edited by U.W. Kulisch and W.L. Miranker, Academic Press (1983).

[3] Neumann, J.v., Goldstine, H.H.: Numerical inverting of matrices of high order, Bull. Amer. Math. Soc., 53 (1947) and Proc. Math. Soc., 2 (1951).