HOW DOES PASCAL-XSC COMPARE TO OTHER PROGRAMMING LANGUAGES WITH RESPECT TO THE IEEE STANDARD?

PASCAL JANSSENS† AND ANNIE CUYT*
DEPARTMENT WIS–INF
UNIVERSITY OF ANTWERP (UIA)
UNIVERSITEITSPLEIN 1
B–2610 WILRIJK-ANTWERP (BELGIUM)

March 1993

ABSTRACT. In this report we investigate a number of popular programming languages, when used for scientific computation or computational mathematics. The main aim is to investigate the XSC-philosophy that has been launched in the last few years and has resulted in a number of XSC-extensions of existing programming languages. We also focus on details of the IEEE standard which, although a standard, is never followed completely.

1. "Genesis"

In the beginning there was a mechanical calculator, for short calculations, limited to a small number of steps, with human supervision. Any difficulties could be noticed as they happened and the accuracy of particular steps adjusted as needed. Then came the computer and lengthy calculations became possible, allowing thousands of millions of steps. Tiny errors, irrelevant in short calculations, could now accumulate and overwhelm the desired answer to the problem. Never in the history of mankind had it been possible to produce so many wrong answers so quickly. Nevertheless, the expectations of the users grew at least as quickly as the power of the computers they were using. In addition, the floating-point calculations went on without supervision and computational crises passed without notice. Of course this created doubts about the reliability of floating-point results. It was clear by now that automatic computation differed substantially from human computation. The discipline of computer arithmetic was born. Although computers were historically developed for scientific computation, extensive interest in the subject of computer arithmetic has mostly developed in the early eighties, yielding some important improvements in the accuracy of floating-point computations when two IEEE floating-point standards were agreed upon [Gol91]. They formalized the representation of the floating-point numbers $I\!F_t \subset I\!R$ and the implementation of the basic floating-point operations (the precision t, carried as a subscript, equals the number of digits in the significant of the float). Among other things [Gol91] these standards specify that the four basic operations $\oplus, \ominus, \otimes$ and $\oslash$ have to be implemented with exact rounding meaning that

$$x \circledast y = \bigcirc(x * y) \qquad * \in \{+, -, \times, /\} \qquad x, y \in I\!F_t$$

where the rounding $\bigcirc : I\!R \rightarrow I\!F$ must be such that if $\bigcirc(r) = f$ then there does not exist another floating-point number between r and f. The exact rounding requirement guarantees maximally accurate results for the arithmetic operations in $I\!F_t$, in the sense that the intermediate result $x * y$ is computed to infinite precision and afterwards rounded to the destination format. However, many computations in numerical algorithms are carried out in product spaces (complex numbers, vectors, matrices, ...). Since the arithmetic operations in these spaces are traditionally performed in terms of the given elementary

Key words and phrases. Scientific Computation, Computer arithmetic, Floating-point numbers.
1993 *Computing Reviews Classification.* G.1.0, D.3, G.4
† email: janssen@wins.uia.ac.be
* Research Director NFWO

floating-point operations in $I\!F_t$, it is easy to see that the computational error due to the accumulation of rounding errors of each of the basic floating-point operations can become quite large. Consider for example the floating-point scalar product $\odot$ of two vectors of floating-point numbers x and y in $VI\!F_t$. Using the basic operations we obtain

$$x \odot y = \bigoplus_{i=1}^{n} x_i \otimes y_i$$

Therefore, a new definition of the basic operations in the product spaces which overcomes this shortcoming, was given in [KM81]. Coming back to our example, the scalar product of two vectors in $VI\!F_t$ is redefined as

$$x \odot y = \bigcirc(x \cdot y) \qquad x, y \in VI\!F_t$$

In [KM81] it is also proven that in order to redefine and implement the basic operations in all these product spaces with exact rounding, the set of basic operations in the IEEE standard only has to be extended with this scalar product for vectors. This extension (X) for scientific (S) computation (C) gave birth to a collection of XSC-languages. Some of these programming languages were offered with a compiler or precompiler (Pascal-XSC), while in other cases libraries were developed as an addition to a popular commercial compiler (C-XSC, Acrith-SC for Fortran).

With the use of supercomputers offering compound operations (like $a \oplus b \otimes c$ or like the scalar product of vectors), institutions like IMACS and GAMM have also expressed the desire that these compound operations be implemented by the manufacturer in such a way that the floating-point result is obtained from the exact result by one single rounding according to the rounding mode in use [MKM92].

2. Reliability of numerical results

In the next sections we screen some compilers available for use on personal computers. We always selected the most recent implementation, for the time being, namely

- the Microsoft Fortran 5.1 compiler, for those still preferring to use the most popular scientific programming language around, namely Fortran;
- the Borland-C^{++} 3.0 compiler, to test a renown product compiling the widely spread and extensively used programming language C^{++};
- The Pascal-XSC 2.02 precompiler to Borland-C^{++} 3.0, for those wanting to experiment with a new product promising reliability and portability

We have chosen these 3 languages because of several reasons.

First of all Fortran has always been the most popular language around when dealing with floats. Nevertheless cases have been reported where an algorithm was executed once in single precision and once in double precision and none of the coinciding digits in both the result types were significant. Knowing that this comparison of precisions is a standard trick for people to test for the error accumulation in their floating-point computations, one must admit that the language lacks tools to guarantee reliability.

Secondly C^{++} as a language for scientific computing has recently been gaining increased attention [Rue92] because of its portability, the class concept and operator overloading that make the language extensible to user-defined structures, and the possibility of more natural interfaces which is also interesting for numerical libraries.

Last but not least XSC-languages have built-in tools for the validation of floating-point computations through their enclosure methods to obtain self-validating numerical algorithms. Interval arithmetic, which has been around for a long time, is the only computational tool so far available that incorporates guarantees as part of the basic computational process, but it has often been criticized since its naive use may deliver bounds which are unreasonably large and thus do not contain much information about the solution of the problem. It is pointed out in [KM86] that this criticism can be superseded. If one combines interval computation, the process of defect correction [SvG89] and the optimal scalar product one can obtain bounds with maximum accuracy. In this approach the role of the optimal scalar product is again crucial.

In order to screen the three selected compilers, we describe how they conform to the IEEE 754 standard for Binary Floating-Point Arithmetic. To comply with this standard a language compiler must provide at least the following 7 items. In the next sections we refer to each of those items with the numbers

given below, preceded by a key depending on the section, namely by MSFT for MS-Fortran 5.1, BCPP for Borland-C^{++} and PXSC for Pascal-XSC 2.02:

(1): a single precision and a double precision floating-point number format, of 32 bits and 64 bits respectively;
(2): 6 basic operations: add, subtract, multiply, divide, square root and remainder, that must be performed with exact rounding;
(3): 4 rounding modes: to nearest, to zero, to $+\infty$ and to $-\infty$;
(4): 5 floating-point exceptions (invalid operation, zero divide, overflow, underflow and inexact) and their handling, including NaN's;
(5): conversions between integer and floating-point formats;
(6): conversions between different floating-point formats;
(7): conversions between basic floating-point numbers and decimal strings.

It's nice that the IEEE standard extended $I\!F_t$ with signed zeroes, denormals, $\pm\infty$ and NaN's, but it would only be truly practical if one could use all these bit patterns as normal floats. However with some of the screened compilers, $\pm\infty$ and NaN's cannot be entered as decimal strings. In MS-Fortran 5.1 and Pascal-XSC 2.02 one has to use the hexadecimal notation instead.

In the sequel of the text, when we refer to a coprocessor, we mean an INTEL coprocessor which was installed complimentary to the INTEL cpu in the personal computer.

3. MS-Fortran 5.1

The Microsoft Fortran Compiler offers a choice of three packages for handling floating-point operations:

The 8087/80287 package: This package allows you to use an 80x87 coprocessor to perform floating-point operations. You must have an 80x87 installed to use this package.

The emulator package: This package uses the 80x87 coprocessor if one is installed. If there is no coprocessor available it provides the necessary 80x87 functions in software. If an 80x87 coprocessor is available you can always force the program to use the emulator by assigning a value to the environment variable NO87. The emulator package should perform basic operations to the same degree of accuracy as a coprocessor. However the emulator routines for transcendental math functions differ slightly from the corresponding 80x87 functions, and this can cause a slight difference (mostly within 2 bits) in the results.

The alternate package: This package gives the smallest and fastest programs you can get without a coprocessor. However, the program results are never as accurate as results from the emulator package. The alternate math package uses a subset of the IEEE standard format numbers: $\pm\infty$, NaN's and denormal numbers are not used. Moreover there are no exception-handlers available in this package.

By default this compiler uses the following mask settings in the control word:

Infinity control	Affine
Round control	Round to Nearest
Precision control	64 bits
Inexact	Masked
Underflow	Masked
Overflow	Unmasked
Zero-divide	Unmasked
Denormalized-operand	Masked
Invalid-operation	Unmasked

These mask settings can be modified by using the procedure `LCWRQQ`. However the user cannot affect the handling of denormals. Retrieval of the status word is done by the procedure `SSWRQQ`. Unfortunately the status word cannot be reset by the user.

To screen this compiler we choose the emulator package because it works both with and without 80x87 coprocessor. Keeping in mind the 7 items we listed in section 2 we now give an overview of the behaviour of floating-points as implemented by the MS-Fortran 5.1 compiler.

(MSFT 1)

Fortran provides the basic floating-point formats as follows :

IEEE Precision	**Fortran**	**range**
Single Precision	real	max_pos = $3.40282347(10^{+38})$
(t = 24)	real*4	min_pos = $1.17549435(10^{-38})$
Double Precision	double precision	max_pos = $1.7976931348623157(10^{+308})$
(t = 53)	real*8	min_pos = $2.22507385072014(10^{-308})$

(MSFT2)

The basic operations $+$, $-$, $\times$ and / are supported for real as well as for double precision floating-point numbers. The remainder and square root are implemented as the intrinsic functions **amod** and **sqrt** for reals and **dmod** and **dsqrt** for double precision numbers. Given $\pm\infty$ or NaN's as argument these last 4 functions produce overflow or domain errors.

Signed zeroes are not supported completely by this compiler: $-0 \times 4 = -0$ but $\sqrt{-0} = +0$ instead of $\sqrt{-0} = -0$ as required. Decimal input of -0 is not possible. Moreover the sign of the result of an operation with $+\infty$ and $-\infty$ is not always correct with the emulator. For example: $+\infty \times (-\infty) = +\infty$.

(MSFT3)

The 4 rounding modes as prescribed by the IEEE standard are available. When using the emulator the extreme case *max_pos + max_pos* delivers NaN in round to zero and round down modes in single precision instead of *max_pos* which should be the result according to the standard. In double precision the results are correct. With the coprocessor rounding works as expected.

(MSFT4)

The Microsoft Fortran 5.1 compiler has three kinds of NaN's: *IND, QNAN* and *SNAN*. The value *IND* is used as the result of invalid operations like 0/0 and $0\times\infty$ and has the bit-pattern of a quiet NaN. However operations with signaling NaN's (*SNAN*) don't result in a quiet NaN. The following operations should all raise the invalid exception and give a quiet NaN as result when the trap is disabled. However from the table below one can see that the IEEE standard is violated in many cases.

	result (with coprocessor)	**raised exception(s) or error**
0/0	IND	Zero-divide and Invalid
$0 \times \pm\infty$	IND	Invalid
$\pm\infty/0$	INF ($\pm\infty$)	Zero-divide
$+\infty - (+\infty)$	IND	Invalid
$\pm\infty/\pm\infty$	IND	Invalid
$\sqrt{-4}$	-	**sqrt**: DOMAIN error
$\sqrt{-4}$	-	**dsqrt**: DOMAIN error
4 REM 0	-	**amod**: DOMAIN error
4 REM 0	-	**dmod**: DOMAIN error

From the table above it is also clear that there are still problems with zero divisors. In the status word the denormal bit is raised from the moment a denormalized number appears in an operation.

(MSFT5)

The rounding of a floating-point number that is converted to an integer is realized by the functions **nint** and **anint**. According to the IEEE standard rounding should be round to even in the default rounding mode but the **nint** function seems to be implemented as:

```
INTEGER FUNCTION NINT(X)
IF (X > 0.0) THEN
    NINT = INT(X+0.5)
ELSE
    NINT = INT(X-0.5)
ENDIF
END
```

which implies that NINT(2.5) = 3.0 instead of 2.0 as expected by the standard. Conversion of *max_pos* to integer raises an invalid operation as expected by the IEEE standard to signal integer overflow, however the result equals 0.

(MSFT6)

Conversion from single precision to double precision works as expected. However the conversion of *max_pos* from double precision to single precision raises an overflow exception and gives $+\infty$ in round up and round to nearest, and NaN in round down and round to zero instead of *max_pos*. The conversion of double precision *min_pos* raises the underflow exception and gives 0 as result in single precision.

(MSFT7)

To test the conversion of decimal strings to floating-point double precision format we consider the following real numbers which we entered specifying all digits (the tests have been performed with and without coprocessor):

$$\begin{aligned}
r_1 &= 1 + 2^{-1} + 2^{-24} + 2^{-53} \in I\!F_{54} \qquad (r_1 \notin I\!F_{53}) \\
&= 1.500000059604644886412927462515654042363166809082031 25 \\
r_2 &= 1 + 2^{-1} + 2^{-24} + 2^{-52} \in I\!F_{53} \\
&= 1.500000059604644997435229925031308084726333618164 0625 \\
\alpha_1 &= 1.50000005960464488 \\
\alpha_2 &= 1.50000005960464490 \\
\alpha_3 &= 1.50000005960464499
\end{aligned}$$

The exact hexadecimal representation of r_1 in double precision is $3FF8\ 0000\ 1000\ 0000\ 8$ which is to be read as

```
sign bit = 0
exponent = 011 1111 1111 (0 biased)
hidden bit = 1
mantissa = 1000 0000 0000 0000 0000 0001 0000
           0000 0000 0000 0000 0000 0000 1000
```

In round up mode this should give us the floating-point number [1]

$$f_{53}^{(u)}(r_1) = 3FF8\ 0000\ 1000\ 0001$$

and in round down mode the floating-point number

$$f_{53}^{(d)}(r_1) = 3FF8\ 0000\ 1000\ 0000$$

The hexadecimal representation of r_2 in double precision is $3FF8\ 0000\ 1000\ 0001$. Since this is a double precision floating-point number we have

$$f_{53}^{(u)}(r_2) = f_{53}^{(d)}(r_2) = r_2$$

Besides, since $r_1 < \alpha_i < r_2$ with $i \in \{2, 3\}$ we must have

$$f_{53}^{(u)}(r_1) \leq f_{53}^{(u)}(\alpha_i) \leq f_{53}^{(u)}(r_2)$$

$$f_{53}^{(d)}(r_1) \leq f_{53}^{(d)}(\alpha_i) \leq f_{53}^{(d)}(r_2)$$

and because there is no $f \in I\!F_{53}$ between $f_{53}^{(d)}(r_1)$ and $f_{53}^{(u)}(r_2) = r_2$ we must also have:

$$f_{53}^{(d)}(\alpha_i) = f_{53}^{(d)}(r_1)$$

and

$$f_{53}^{(u)}(\alpha_i) = f_{53}^{(u)}(r_2) = r_2$$

Unfortunately in MS Fortran 5.1 we get

$$f_{53}^{(u)}(r_1) = f_{53}^{(d)}(r_1) = 3FF8\ 0000\ 1000\ 0000$$

[1] upperscript indices indicate rounding mode

which is wrong, and for the rest

$$f_{53}^{(u)}(r_2) = f_{53}^{(d)}(r_2) = r_2$$
$$f_{53}^{(d)}(\alpha_1) = 3FF8\ 0000\ 1000\ 0000 = f_{53}^{(d)}(r_1)$$
$$f_{53}^{(d)}(\alpha_2) = 3FF8\ 0000\ 1000\ 0001 = r_2 \neq f_{53}^{(d)}(r_1)$$
$$f_{53}^{(d)}(\alpha_3) = 3FF8\ 0000\ 1000\ 0001 = r_2 \neq f_{53}^{(d)}(r_1)$$
$$f_{53}^{(u)}(\alpha_1) = 3FF8\ 0000\ 1000\ 0000 \neq r_2$$
$$f_{53}^{(u)}(\alpha_2) = 3FF8\ 0000\ 1000\ 0001 = r_2$$
$$f_{53}^{(u)}(\alpha_3) = 3FF8\ 0000\ 1000\ 0001 = r_2$$

Since $\alpha_i < r_2$, these results imply that for $i \in \{2, 3\}$

$$r_2 = f_{53}^{(d)}(\alpha_i) > \alpha_i$$

which is nonsense.

4. Borland-C^{++} 3.0

As for MS-Fortran, programs in Borland-C^{++} can be compiled in four ways regarding floating-point operations:

None: In this mode you can't do any floating-point operations.
Emulator: In this mode the program detects whether your computer has an 80x87 coprocessor and will use it if it's there. If there is no coprocessor detected your program will emulate one.
8087: In this mode direct 8087 coprocessor inline code is used.
80287: In this mode direct 80287 coprocessor inline code is used for floating-point operations.

In some situations you might want to override the default 80x87 auto-detection behavior. For example when you want to know if your program also produces the right results on machines without a coprocessor. With the 87 environment variable you can modify this behavior. Setting the 87 environment variable to *N* (for No) with the command "SET 87=N" at the DOS prompt tells the startup code that you don't want to use the 80x87, even when it is available. Setting this environment variable to *Y* (for Yes) means that the coprocessor is there, and you want the program to use it. If you set the 87 environment variable to *Y* in absence of a coprocessor your system will hang.

The default settings for the control word are as follows:

Infinity control	Affine
Round control	Round to Nearest
Precision control	64 bits
Inexact	Masked
Underflow	Masked
Overflow	Unmasked
Zero-divide	Unmasked
Denormalized-operand	Masked
Invalid-operation	Unmasked

By default Borland-C^{++} programs abort if a floating-point overflow, divide by zero or invalid error occurs. This behavior can be changed by masking floating-point exceptions by calls to `_control87`. You can determine whether a floating-point exception occurred by calling `_status87` or `_clear87` after its occurrence. The function `_clear87` clears the status word. Certain math errors can also occur in library functions. For example: $\sqrt{-4}$ prints an error message on the screen and returns a NaN.

Let us now screen the implementation of floating-points with respect to the IEEE standard discussing the 7 items of section 2.

(BCPP1)

The following floating-point formats are provided (since long double is not a required precision of the standard, we will not mention it further in our discussion of Borland-C^{++}):

IEEE Precision	C
Single Precision (t=24)	float
Double Precision (t=53)	double
Extended Precision (t=64)	long double

C	size	range
float	32 bits	$1.17549435(10^{-38})$ to $3.40282347(10^{+38})$
double	64 bits	$2.2250738585072014(10^{-308})$ to $1.7976931348623157(10^{+308})$
long double	80 bits	$3.362103143112093506262677817321752 6(10^{-4932})$ to $1.189731495357231765085759326628007 0(10^{+4932})$

(BCPP2)

The basic operations $+, -, \times$ and / are available for the two required precisions. The remainder and the square root are only defined for double precision arguments and give a double precision number as result. Borland-C^{++} does not issue an error when one calls these two functions with a single precision argument, but it gives wrong results and continues with the program. For example: $\sqrt{9}$ gives $-3.27(10^{+4})$.

Borland-C^{++} supports signed zeroes in all its operations. Furthermore $\sqrt{-0} = -0$. Operations with denormal numbers are only possible with a coprocessor installed. The emulator flushes all the denormals to zero.

(BCPP3)

The 4 rounding modes (round to nearest, round up, round down and round to zero) are supported for both precisions and act as expected by the IEEE standard.

(BCPP4)

The denormal bit of the status word is not used in Borland-C^{++}. Due to the fact that with the emulator all denormals are flushed to zero, the zero divide exception is raised when dividing by a denormal number. Using a coprocessor results in correct behavior.

The handling of invalid operations when the invalid trap is disabled, is summarized in the following table. The invalid operations of the remainder function deliver 0 instead of NaN and don't raise the invalid exception.

	result	raised exception or error
$0/0$	$+\infty$	Zero-divide (without 80x87)
$0/0$	NaN	Invalid (with 80x87)
$0 \times \pm\infty$	NaN	Invalid
$\pm\infty/0$	$\pm\infty$	-
$+\infty - (+\infty)$	NaN	Invalid
$\pm\infty/\pm\infty$	NaN	Invalid
$\sqrt{-4}$	NaN	`sqrt`: DOMAIN error
4 REM 0	0	-
$+\infty$ REM 5	0	-

(BCPP5)

As for the screened Fortran compiler, Borland-C^{++} delivers a zero result when converting *max_pos* to an integer value. To signal the integer overflow the invalid exception is raised.

(BCPP6)

Conversion from single precision to double precision works as expected. Conversion of *max_pos* from double precision to single precision raises an overflow exception and gives $+\infty$ in round up and round to nearest while returning single precision *max_pos* in the other two rounding modes. This is the correct behavior according to the IEEE standard. Conversion of double precision *min_pos* raises the underflow exception and returns 0 regardless of the rounding mode.

(BCPP7)

The conversion of decimal strings to floating-point double precision format is tested using the input of the previous section. We obtain the following results for $i \in \{1, 2, 3\}$:

$$f_{53}^{(d)}(r_1) = f_{53}^{(d)}(r_2) = f_{53}^{(d)}(\alpha_i) = 3FF8\ 0000\ 1000\ 0000$$

$$f_{53}^{(u)}(r_1) = f_{53}^{(u)}(r_2) = f_{53}^{(u)}(\alpha_i) = 3FF8\ 0000\ 1000\ 0000$$

Consequently we have

$$f_{53}^{(d)}(r_2) \neq f_{53}^{(u)}(r_2) = r_2$$

which is wrong since $r_2 \in I\!F_{53}$. The other equations are respected by Borland-C^{++} since all downward roundings and upward roundings are equal respectively.

5. Pascal-XSC 2.02

The software arithmetic of Pascal-XSC is realized using integer arithmetic. In case an IEEE arithmetic coprocessor is available, it can be used. However, a software emulation of the optimal dot product for accumulation of numbers and products is still necessary to obtain reliable results. In contrast to usual programming languages like Fortran, Pascal and Modula 2, Pascal-XSC provides all operations in product spaces like $V\!I\!R$ (real vectors), $I\!I\!R$ (real intervals) and $M\!I\!R$ (real matrices) via the usual operator symbols. Each of these operations calls elementary operations which are implemented using the optimal dot product. Moreover Pascal-XSC has explicit language support for directed roundings (downward ∇ and upward Δ) and the corresponding operations ($\triangledown$ and $\triangle$ for all $* \in \{+,-,\times,/\}$).

By default, the exception handlers for the exceptions division by zero, exponent overflow and invalid operation are enabled and cause the termination of the program after error messages are displayed. The runtime option **-ieee** is provided for changing the default settings of the enabled status of exception handlers. Another possibility of changing the status of the exception handling environment for IEEE exception is given by the use of the procedures **IEEE_trap_enable** and **IEEE_environment**. With the procedure **IEEE_environment** you decide whether the processing of the program has to be continued after leaving a trap-handler. The procedure **IEEE_trap_enable** lets you change the state of the trap-handlers. Using the procedure **IEEE_test** one can test if a particular exception has occurred during the exécution of an operation. With the procedure **IEEE_reset** the status word can be reset.

(PXSC1)

Pascal-XSC only supports the double precision floating-point format (t=53). However, it is called real. The real normal floating-point numbers range from 2.2250738585072013(10^{-308}) to 1.7976931348623158 (10^{+308}).

(PXSC2)

The basic operations $+,-,\times,/$ and the square root are supported for double precision numbers. The remainder function, as required by the IEEE standard, however is not provided. Signed zeroes are available in Pascal-XSC, but if you want them displayed in decimal notation you will have to start your program with the command-line option **-sz**. $\sqrt{-0} = +0$ instead of $\sqrt{-0} = -0$ as required. As for Borland-C^{++}, operations with denormal operands are only possible if you have a coprocessor installed. When there's no coprocessor denormal numbers are flushed to zero. Operations with $\pm\infty$ are not always correct regarding the sign of the result: $+\infty \times (-\infty) = +\infty$.

(PXSC3)

Global rounding modes fixed by a bit-pattern in the control word, as used in C and Fortran, are not available in Pascal-XSC. One reason for this, is the fact that some arithmetic functions reset the value of the rounding bits of the control word to its default (round to nearest). Further operations are then fulfilled in round to nearest mode, which can give unexpected results. Therefore Pascal-XSC provides each basic operation, even **read** and **write**, in three rounding modes. Rounding towards zero is not yet provided.

(PXSC4)

Regardless of the invalid trap handler $\sqrt{-4}$ gives the error message "ieee math error: -Normal in function sqrt". The other invalid operations are handled in the following way by Pascal-XSC:

	result	exception	error when trap is enabled
$0/0$	$1.335045212497809(10^{-306})$	Invalid	invalid operation: 0 / 0
$0 \times \pm\infty$	$\pm\infty$	Invalid	invalid operation: 0 × infinity
$\pm\infty/0$	$\pm\infty$	-	
$+\infty - (+\infty)$	$+\infty$	Invalid	unexpected infinity operand
$\pm\infty/\pm\infty$	$\pm\infty$	Invalid	
$\sqrt{-4}$	-	-	

(PXSC5)

Conversion from double precision *max_pos* to integer raises an integer overflow error and delivers 0 as result.

(PXSC6)

Conversion between different floating-point formats is not possible since there is only one format provided, namely the double precision.

(PXSC7)

Recalling the definitions of r_1, r_2, α_i where $i \in \{1,2,3\}$ we obtain the following results in Pascal-XSC:

$$\begin{aligned} f_{53}^{(d)}(r_1) &= 3FF8\ 0000\ 1000\ 0000 \\ f_{53}^{(u)}(r_1) &= 3FF8\ 0000\ 1000\ 0001 \\ f_{53}^{(d)}(r_2) &= f_{53}^{(u)}(r_2) = 3FF8\ 0000\ 1000\ 0001 = r_2 \\ f_{53}^{(d)}(\alpha_i) &= 3FF8\ 0000\ 1000\ 0000 = f_{53}^{(d)}(r_1) \\ f_{53}^{(u)}(\alpha_i) &= 3FF8\ 0000\ 1000\ 0000 = r_2 \end{aligned}$$

with $i \in \{1,2,3\}$.
Let us also consider the following real number:

$$r_3 = 1 + 2^{-1} + 2^{-24} + 2^{-52} + 2^{-80} = r_2 + 2^{-80}$$

Pascal-XSC gives the following result (the entire decimal string was entered and apparently also processed):

$$\begin{aligned} f_{53}^{(d)}(r_3) &= 3FF8\ 0000\ 1000\ 0000 \\ f_{53}^{(u)}(r_3) &= 3FF8\ 0000\ 1000\ 0001 \end{aligned}$$

We can easily conclude from these results that Pascal-XSC supports the conversion between decimal and binary representation as required by the IEEE standard. It even detects a bit way out in the real number: $2^{-80} \approx 10^{-24}$. The programmer really feels in control: if he or she wants to enter a long number, all the digits are respected. If such precise input is not necessary, the programmer enters less digits. Anyway, from (MSFT7) and (BCPP7) we see that even exact floating-point numbers belonging to $I\!F_{53}$ cannot be retrieved with these compilers.

6. Dealing with cancellation in floating-point computations

Let us consider the following problem involving floating-point computations. We want to evaluate the polynomial

$$\begin{aligned} p(x) &= a_3x^3 + \ldots + a_0 \\ &= 543339720x^3 - 768398401x^2 - 1086679440x + 1536796802 \end{aligned}$$

in $x = 1.414215087890625$ by means of the Horner scheme:

$$\begin{aligned} p(x) &:= a_3 \\ p(x) &:= p(x) \otimes x \oplus a_{3-i} \qquad i = 1, \ldots, 3 \end{aligned}$$

The correct value is

$$p(x) = 3.57644118525968224\ldots(10^{-3})$$

Although this problem is algebraically simple, it is computationally hard. We are facing catastrophic cancellation in the Horner scheme as well as in a straightforward evaluation. This difficulty is further discussed in the next section. Let us first list the different numerical results for the different compilers. With MS-Fortran and Borland-C^{++} we obtain (with and without coprocessor):

$$p(x) = 3.5766\,(10^{-3})$$

Notice that we only print the significant digits because we know the correct result. In real-life situations this is in general not possible. With Pascal-XSC we obtain

$$p(x) \in [3.576441185259681\,(10^{-3}), 3.576441185259683\,(10^{-3})]$$

Here the number of significant digits can be read immediately from the interval output.

7. Conclusions

With the end of the 20^{th} century and with the great era of computers in view, scientists do by now expect to have a reliable "calculator" at their disposal. Nothing is less true.

The idea to complement computations with an error analysis through interval arithmetic combined with defect correction, is a good step in that direction. It is the XSC-philosophy. However, using a long accumulator for compound operations does not necessarily improve the results. Even when the accumulator is filled with only slightly contaminated numbers, the error can spread. Consider for instance the expression

$$y = 27a^6 - 10a^3b^3 - b^6 - 3ab - 12a^2b^2$$

with $a = 12970$ and $b = 16897$. For this integer input stored in double precision variables, everything is of course in order because when calculating y in a long accumulator nor a, nor b nor any of the intermediate results have to be rounded. All the outcoming digits are reliable. In case of cancellation all the remaining digits are correct (by the way $y = 1$ which is almost never computable).

Taking floating-point input $a = 1297.0$ and $b = 1689.7$, the best we can do for b always differs from its exact value in at least the 53^{rd} bit. Hence

$$\bigcirc(b) = b + \epsilon$$
$$|\epsilon| \leq 2\,(10^{-12})$$

Once this last digit is contaminated, up to now nothing can prevent the catastrophic effect when computing b^6:

$$b^6 \approx \bigcirc(b) \otimes \ldots \otimes \bigcirc(b) = (b+\epsilon) \otimes \ldots \otimes (b+\epsilon) \approx b^6 + 6b^5\epsilon + \ldots$$
$$|6b^5\epsilon| \leq 1\,(10^8)$$

In case of cancellation in a long accumulator, this error term will now remain as the dominant result.

In order for this to be taken care of, extra precision for b should be provided dynamically, after the catastrophic effect being signaled for instance by reliable error control methods. It is our intention that future research be carried out for that type of problems.

References

[Gol91] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comp. Surv. **23** (1991), 5–48.

[KM81] U. Kulisch and W. Miranker, *Computer arithmetic in theory and practice*, Academic Press, New York, 1981.

[KM86] U. Kulish and W. Miranker, *The arithmetic of the digital computer : A new approach*, SIAM Review **28** (1986), 1–36.

[MKM92] S. M. Markov, E. Kaucher, and G. Mayer (eds.), *Computer arithmetic, scientific computation and mathematical modelling*, Baltzer, Basel, 1992.

[Rue92] U. Ruede, *Synopsis on a workshop on scientific computing in c^{++}*, Tech. report, Report T.U. München, 1992.

[SvG89] G. Schumacher and J. Wolff von Gudenberg, *Highly accurate numerical algorithms*, In Ullrich and von Gudenberg [UvG89], pp. 1–17.

[UvG89] Ch. Ullrich and J. Wolff von Gudenberg (eds.), *Accurate numerical algorithms, a collection of research papers*, Berlin, Springer-Verlag, 1989.