TESTING THE NUMERICAL PRECISIONS REQUIRED TO EXECUTE REAL WORLD PROGRAMS

John Collins

Remote Radar Sensing Group, University of Cape Town, South Africa
Department of Computing, Edge Hill University, UK
SimCon Ltd. UK.

ABSTRACT

The IEEE 754 Standard for floating point number formats is over 30 years old. Computational requirements have changed dramatically during these 30 years. This paper describes methods by which, for a given program required to yield results of a given accuracy, one may determine:

1) *whether the choices of numerical precision are adequate;*
2) *the minimum numerical precisions actually required;*
3) *the statements most vulnerable to loss of precision.*

Precision is specified as an exact number of mantissa bits for each real kind.

The method is applied to two small Fortran simulation programs with surprising results. The implications are that considerable savings could be made in computer hardware by computing with reduced precision, and that speed improvements may be made on existing systems by packing data for inter-processor communication, removing those parts of the bit patterns which are not needed.

KEYWORDS

Numerical precision, automated software engineering, emulation of arithmetic

1. INTRODUCTION

1.1. Motivation

This study was carried out for two reasons.

a) To test the adequacy of the choices of precision in existing programs

Most modern large programs use two, or sometimes three of the real number precisions defined in the IEEE 754 Standard [1]. The analysis of large programs in aerospace technology and climate research has revealed very large numbers of inconsistencies in the precisions chosen. For example, the major weather forecasting and climate code WRF [2] contains over 4,700 occurrences of mixed precision in arithmetic statements, some of which are expected to cause problems (Collins et al [2011] [3], [2012] [4] and [2013] [5]). The methods described in this paper test for the adequacy of the choice of precision, and identify the statements where the precision is too low.

b) To investigate possible speed improvements in high performance computing

There is a trade-off between arithmetic speed and computational precision. Typically, for example, addition or multiplication of 32-bit IEEE 754 real numbers is about twice as fast as the corresponding operations on 64-bit numbers. There are also significant savings in cache coherence and inter-processor communication where lower precisions can be used, because the numbers occupy less memory. Düben and Palmer [2014] [6], Düben et al [2015] [7] and Dawson and Düben, [2016] [8] tested the use of reduced precision numbers in small atmospheric codes, and found that the performance is resilient to reductions in precision significantly below the 23 mantissa bits in 32-bit IEEE 754 numbers. The urgent need and economic justification for greatly increased computing power in weather and climate prediction is well described by Shukla et al [2010] [9]. The precise measurement of the precisions required in a program may make it possible to build hardware, compilers and procedures to take advantage of reduced precision computing as a step towards significantly improved computing power
.

1.2. Successful Use of Lower Precisions: The AD10

The most important real-time design simulations of the space shuttle, the launch, main engine and robot arm, were made with 16-bit integer arithmetic on a fixed-point machine. The same technology was used to simulate nuclear power plant (Cheng et al [1984] [10]), aircraft engines (Peterson [1984] [11]), car power-train (Hogh et al [1988] [12]) and major missile systems (e.g. Hanson [1984] [13] and Johnson [1987] [14]). The machine used was the Applied Dynamics International AD10 [15]. All derivative computations were made as fixed point 16-bit scaled fractions, and the integrals, held in a specialised numerical integration processor, were stored as 48-bit scaled fractions but were reduced to 16-bit values when they were exported for use in derivative computations. 16-bit values, with 48-bits for the integrals, were entirely adequate for a wide range of engineering simulations.

1.3. Scale Separation

Note that the AD10 used 48-bit scaled fractions for integrals. It was recognised that some computations, in particular numerical integration, could not be performed adequately with low precision variables. This effect has been described in recent experiments in reduced precision computing, for example, by Düben, McNamara and Palmer [2014] [16]. It is important, in setting up a reduced precision regime, to identify those computations which must be carried out at higher precisions, and the methods described in this paper provide this facility.

1.4. Specifying Precision

In this study, the precisions of real numbers are specified to an exact number of mantissa bits. The numbers are not restricted to the numbers specified in the IEEE 754 standard and may take any value less than or equal to the number of mantissa bits in the variables in the original program.

2. TESTING REAL PRECISION: THE APPROACH

2.1. Systematic Changes to Declarations

The method described is applied to Fortran programs (The presence of components written in C or other languages does not present a problem).

All REAL and COMPLEX variables and Fortran parameters (Named constants) in the program under study are changed to objects of derived types which emulate the real numbers. This change is made automatically by a software engineering tool, and affects the declarations of the variables and a very small number of special cases in the executable code. For example, the declaration:

```
        REAL :: x
```

is changed to:

```
        TYPE (em_real_k4) :: x
```

Note that the original declaration may be written in many different ways and with arbitrary use of white space. The software engineering tool must recognise all equivalent declarations and make the appropriate change. The software tool used, fpt [17], contains a full static semantic analyser. It recognises all forms of REAL and COMPLEX declarations and automatically adds declarations for implicitly typed objects.

The emulated real and complex types each contain a single component of the corresponding native type. Thus, for example, the type **`em_real_k4`** contains a component of the native type **`REAL(KIND=kr4)`** where the integer **`kr4`** is the kind value for a 32-bit IEEE 754 number. The emulated types used in the present study are:

```
!       Note that all types are sequence derived types so that
!       the variables can be in COMMON

        TYPE em_real_k4
           SEQUENCE
           REAL(KIND=kr4) value
        END TYPE em_real_k4

        TYPE em_real_k8
           SEQUENCE
           REAL(KIND=kr8) value
        END TYPE em_real_k8

        TYPE em_complex_k4
           SEQUENCE
           COMPLEX(KIND=kr4) value
        END TYPE em_complex_k4

        TYPE em_complex_k8
           SEQUENCE
           COMPLEX(KIND=kr8) value
        END TYPE em_complex_k8

        PUBLIC em_real_k4
        PUBLIC em_real_k8
        PUBLIC em_complex_k4
        PUBLIC em_complex_k8
```

2.2. Overloading Arithmetic Operators

The program code contains arithmetic expressions, for example, **`a = b * c`**. The compiler has no rule to specify what the arithmetic operator **`*`** is to do with objects of the derived type **`em_real_k4.`** The rules are specified by overloading the arithmetic operators. The overloads are written in a Fortran module which is referenced in every top-level compilation unit. The reference is made by inserting the statement:

```
        USE module_emulate_real_arithmetic
```

The insertion is made automatically by the software engineering tool. Each overload is made by calling a Fortran subroutine which carries out the overloaded operation.

The changes in precision are made in the functions and subroutines which overload the arithmetic. For example, the function which overloads multiplication of two emulated 32-bit IEEE 754 real variables is:

```
ELEMENTAL FUNCTION multiply_em_real_k4_em_real_k4(a1mr4,a2mr4)
        TYPE (em_real_k4) :: multiply_em_real_k4_em_real_k4
        TYPE (em_real_k4),INTENT(IN) :: a1mr4
        TYPE (em_real_k4),INTENT(IN) :: a2mr4
        TYPE (em_real_k4) :: t1mr4
        TYPE (em_real_k4) :: t2mr4
        t1mr4 = experiment_4(a1mr4)
        t2mr4 = experiment_4(a2mr4)
        multiply_em_real_k4_em_real_k4%value = &
         & t1mr4%value * t2mr4%value
        multiply_em_real_k4_em_real_k4 = &
         & experiment_4(multiply_em_real_k4_em_real_k4)
END FUNCTION multiply_em_real_k4_em_real_k4
```

Within each function or subroutine which overloads the arithmetic, the change in precision is made by calls to one of the functions **`experiment_4`** for 32-bit real numbers, **`experiment_8`** for 64-bit real numbers or by **`experiment_x4`** or **`experiment_x8`** for the corresponding complex numbers. These functions round the mantissa of the emulated real number to a specified number of bits. Note that the functions round the arguments. They do not simply truncate them because this would cause a systematic bias in the result. The rounding function is called both before and after each arithmetic operation to prevent additional precision in values from I/O statements or constants in the code from entering the computations. This strategy allows literal values in the code and I/O statements to remain unchanged.

The number of mantissa bits for each real kind, for example **`kr4_mantissa_bits`** for 32-bit numbers, is read from a file at the start of the program. A call to an initialisation subroutine is automatically inserted by the software engineering tool. In the interest of efficiency, the functions which change the precision transfer the bit patterns of the real or complex numbers to integers, mask off the exponents and adjust the mantissae. The control variables for the routines which change the precision of 32-bit numbers are:

```
        INTEGER(KIND=ki4),PARAMETER :: &
         kr4_exponent_lsb = Z'00800000'
        INTEGER(KIND=ki4),PARAMETER :: &
         kr4_exponent_mask = Z'FF800000'
        INTEGER :: kr4_mantissa_bits  ! Read from file

        INTEGER(KIND=ki4) :: kr4_test_bit
        INTEGER(KIND=ki4) :: kr4_test_field
        INTEGER(KIND=ki4) :: kr4_mask
        INTEGER(KIND=ki4) :: kr4_max_mantissa
```

The control variables are initialised when the program starts for each test case of precision, e.g. for 32-bit numbers:

```
        kr4_test_bit = ISHFT(1_ki4,(22-kr4_mantissa_bits))
        kr4_test_field = kr4_test_bit-1
        kr4_mask = &
         IEOR(-1_ki4,ISHFT(1_ki4,(23-kr4_mantissa_bits))-1)
        kr4_max_mantissa = &
         IAND(Z'004FFFFF',(kr4_mask+kr4_test_bit))
```

The function to round a 32-bit number to the chosen number of mantissa bits is:

```
ELEMENTAL FUNCTION experiment_4(r)
        IMPLICIT NONE
        TYPE(em_real_k4),INTENT(IN) :: r
        REAL(KIND=kr4) :: experiment_4
        INTEGER(KIND=ki4)ir
        ir = TRANSFER(r,ir)
        IF (IAND(ir,kr4_test_bit) /= 0) THEN
           IF (IAND(ir,kr4_test_field) /= 0) THEN
              IF (IAND(ir,kr4_max_mantissa) == &
                  kr4_max_mantissa) THEN
                 ir = IAND(ir,kr4_exponent_mask) &
                  + kr4_exponent_lsb
              ELSE
                 ir = ir+kr4_test_bit
              ENDIF
           ENDIF
        ENDIF
        ir = IAND(ir,kr4_mask)
        experiment_4 = TRANSFER(ir,experiment_4)

END FUNCTION experiment_4
```

Additional functions could easily be added to round 80 and 128-bit real numbers.

2.3. The Fortran Module `module_emulate_real_arithmetic`

This module contains the routines described above to round numbers to the required precision, and all of the arithmetic and intrinsic function overloads. The module contains a large number of sub-programs. Sub-programs are needed to overload:

1) simple assignment;
2) the unary operators `+` and `-`;
3) the arithmetic operators `+`, `-`, `*`, `/` and `**`;

4) the relational operators **==**, **/=**, **>**, **<**, **>=** and **<=**;
5) all intrinsic functions and subroutines with real or complex arguments.

For the dyadic cases of operators, these must be supplied for all combinations of emulated 32-bit and 64-bit real variables and the corresponding complex variables, with emulated 32-bit and 64-bit real variables and the corresponding complex variables, with 32-bit and 64-bit real literal numbers and the corresponding complex literal numbers, and with 8, 16, 32 and 64-bit integer objects. The codes of many of these sub-programs are closely similar, and programs were written to generate them automatically. Dawson and Düben [2016] [8] developed a similar Fortran module for emulation of reduced precision arithmetic.

2.4. Additional Program Changes - Special Cases in Emulating Real Arithmetic

There are a number of additional changes which must be made to a program when the arithmetic is emulated. Most of these, except for the changes to parameter and data specifications, are infrequent, but they must be handled by the software engineering tools if the entire process is to be automatic. They are:

2.4.1. Changes to the values specified for Fortran parameters

Fortran parameters are compile-time constants. Real and complex Fortran parameters are converted to objects of the corresponding emulated types. The declaration of a Fortran parameter might be written, for example:

```
REAL(KIND=4),PARAMETER :: c = 2.9979E+8
```

When the real objects are emulated, it is not sufficient simply to change the declaration to:

```
TYPE (em_real_k4),PARAMETER :: c = 2.9979E+8
```

The problem is that the conversion of the real value **2.9979E+8** to an emulated 32-bit real number has been defined by a subroutine, but the compiler cannot call this subroutine at compile time. The value **2.9979E+8** must be explicitly converted to the emulated type. The re-engineered code must be written:

```
TYPE (em_real_k4),PARAMETER :: c = em_real_k4(2.9979E+8)
```

In the same way, the functions for arithmetic operators and intrinsic functions cannot be used to compute values at compile time. Therefore, for example, the statements in the original code:

```
REAL (KIND=kr4),PARAMETER :: pi = 3.141593
REAL (KIND=kr4),PARAMETER :: tpi = 2 * pi
```

must be converted to:

```
TYPE (em_real_k4),PARAMETER :: pi = em_real_k4(3.141593)
TYPE (em_real_k4),PARAMETER :: tpi = &
 em_real_k4( 2 * pi%value )
```

2.4.2. Changes to real and complex data specifications

The values specified in data statements and in embedded data in declarations are also compile-time constants. The compiler cannot call the subroutines which overload real and complex assignments, so the values must be converted in the same way as in the PARAMETER statements shown above.

2.4.3. Intrinsic functions which may convert the kind of the result

Some Fortran intrinsic functions have an argument which specifies the kind of the real or complex value returned. The author has not found a reliable way to implement a user-written function which returns a value of a kind which depends on the function arguments. The engineering tool is therefore used to replace these function invocations, either using the obsolescent intrinsics **`DBLE`** and **`SNGL`** or by a two stage process. For example, where **`r8`** is an emulated 64-bit real variable, and **`i8`** is a 64-bit integer variable, the code:

```
r8 = REAL(i8,8)
```

is replaced by

```
r8 = DBLE(i8)
```

The code

```
i8 = INT(r8,8)
```

is replaced by

```
i8 = INT(r8%value,8)
```

2.4.4. The intrinsics `MAX` and `MIN`

The intrinsic functions **`MAX`** and **`MIN`** may have an arbitrary number of arguments, and most Fortran compilers allow the arguments to be of different kinds, and sometimes of different data types. This leads to a factorial explosion in the number of overload functions required to emulate them.

The code for invocations of **`MAX`** and **`MIN`**, where at least one of the arguments is real, is therefore modified as follows:

1) A check is made that no more than one of the arguments is a literal real value. If two or more arguments are literal reals, the **`MAX`** or **`MIN`** of the literals is computed and the appropriate value is left in place. The others are removed.
2) The kind of every argument is examined, and all of the arguments are converted to real objects of the kind with the highest precision. This is the observed behaviour of the compilers which have been tested. The return value is of this kind.

Thus, if **`r8`** is a 64-bit real variable and **`r4`** is a 32-bit real variable, the statement:

```
r8 = MAX(r4, 1, r8)
```

is automatically replaced by

```
r8 = MAX(DBLE(r4), em_real_k8(1.0D0), r8)
```

r4 becomes an emulated 32-bit variable**,** **r8** becomes an emulated 64-bit variable, **DBLE** is overloaded and the literal value **1** is converted to an emulated 64-bit value.

2.5. Statements and Constructs which Do Not Need to be Changed

The derived types used to emulate the real numbers are **SEQUENCE** derived types which contain only a single component of the original real or complex kind. As a result, the following constructs do not need to be changed:

1) I/O statements, e.g. **READ, WRITE, ACCEPT** etc.;
2) Literal constants in the code. As soon as a constant is used in any arithmetic operation it is converted to the corresponding derived type with the specified precision before it is used;
3) **EQUIVALENCE** statements and **COMMON** blocks which depend on the detailed organisation of memory: note that **SEQUENCE** derived types may be placed in **COMMON** blocks. **COMMON** and **EQUIVALENCE** are disappearing from the Fortran standard, but they have not yet disappeared from existing Fortran programs;
4) Calls to sub-programs written in C or other languages: the single-component derived types are passed in exactly the same way as the original real variables.

Dawson and Düben, [2016] [8], independently created a similar Fortran module to emulate real arithmetic with reduced precision. However, they used derived types with multiple components. The advantage of their approach is that it is possible to use different precisions for different objects of the same real kind. The disadvantage is that many more changes must be made to an existing program.

3. CHANGING A PROGRAM TO TEST PRECISION

The steps required to change an existing Fortran program to test the adequacy of the chosen precisions are:

1) Insert a USE statement for the emulation module in every top-level program unit;
2) Insert a call to the routine which initialises the emulation system at the start of each main program (Usually there is only one);
3) Ensure that all real and complex objects are declared;
4) Convert all real and complex declarations to the emulated derived type declarations;
5) Carry out any remaining systematic changes required to handle the derived types;
6) Build the re-engineered program;
7) The number of bits of precision in each emulated REAL kind are specified in a text file which is read when the emulation code is initialised. Edit this file to specify the required precisions;
8) Run the code, and assess the adequacy of the results.

Repeat steps 7) and 8) until the lowest safe precision has been identified. If the results change when the precision is reduced by only a single bit from the native value, the native precision is probably already inadequate.

In this study, the first 5 steps are carried out by the software engineering tool in a single pass. Note that the last two steps may be repeated without re-creating or rebuilding the re-engineered code.

4. APPLICATION TO EXAMPLE CODES

4.1. A First Example: met_rocket.f

The program **met_rocket.f** is a simulation of a two-stage meteorological rocket. It was written at the Overberg Test Range in South Africa to address a real engineering problem.

Extracts from the code are shown below. The complete code is available at [18]:

```
PROGRAM met_rocket
!
        IMPLICIT NONE
!
! **************************************************************
!
        REAL                                                    &
         g                                                      &
         ,p_burn_time                                           &
         ,p_burn_thrust                                         &
         ,p_initial_mass                                        &
         ,p_stage_2_mass                                        &
         ,p_fuel_mass                                           &
         ,p_launch_vel                                          &
         ,frametime
        PARAMETER (                                             &
         g = 32.2D0                    & ! fpsps
         ,p_burn_time = 2.11D0         & ! secs
         ,p_burn_thrust = 4021.0D0*g   & ! lb-ft
         ,p_initial_mass = 67.2D0      & ! lb
         ,p_stage_2_mass = 10.0D0      & ! lb
         ,p_fuel_mass = 37.6D0         & ! lb
         ,p_launch_vel = 190.0         & ! fps
         ,frametime = 0.001D0          & ! secs
         )
```

(Integer declarations removed)

```
        REAL                                                    &
         dtr                                                    &
         ,time                                                  &
         ,xdd               & ! Down range acceleration, fpsps
         ,xd                & ! Down range velocity, f/s
         ,x                 & ! Down range distance, ft
         ,pxdd              & ! Last frame accelleration
         ,pxd               & ! Last frame velocity
```

Note that the real variables in this simulation are all declared to be of the default real kind. They are 32-bit reals. This declaration was almost certainly accidental. It would be normal to use 64-bit variables in a simulation of this type. The literal real values used to initialise the simulation are 64-bit numbers, but the additional precision of these numbers is lost at once.

When this code is run under gfortran the rocket reaches a maximum altitude of 237,446 feet.

The **met_rocket** program was re-engineered to emulate the real arithmetic. The corresponding sections of the re-engineered code are shown below:

```
PROGRAM met_rocket
!
        USE module_emulate_real_arithmetic
!
        IMPLICIT NONE
!
! **************************************************************
!
        TYPE (em_real_k4)                                       &
         g                                                      &
         ,p_burn_time                                           &
         ,p_burn_thrust                                         &
         ,p_initial_mass                                        &
         ,p_stage_2_mass                                        &
         ,p_fuel_mass                                           &
         ,p_launch_vel                                          &
         ,frametime
        PARAMETER (                                             &
         g = em_real_k4( 0.32200000000000028E+02) &! fpsps
         ,p_burn_time = em_real_k4(0.21099999999999988E+01)  &
         ,p_burn_thrust = em_real_k4(0.12947620000000001E+06) &
         ,p_initial_mass = em_real_k4(0.67200000000000028E+02) &
         ,p_stage_2_mass = em_real_k4( 0.1E+02) &! lb
         ,p_fuel_mass = em_real_k4( 0.37600000000000014E+02) &
         ,p_launch_vel = em_real_k4(190.0) &! fps
         ,frametime = em_real_k4( 0.10000000000000002E-02)   &
         )
```

Integer declarations removed

```
        TYPE (em_real_k4)                                       &
         dtr                                                    &
         ,time                                                  &
         ,xdd                & ! Down range acceleration, fpsps
         ,xd                 & ! Down range velocity, f/s
         ,x                  & ! Down range distance, ft
         ,pxdd               & ! Last frame accelleration
         ,pxd                & ! Last frame velocity
```

Note that REAL declarations have been changed and that PARAMETER statements have been re-engineered. The only other change to the entire program is the insertion of a call to the subroutine **initialise_emulated_arithmetic** as the first executable statement.

The first test of the re-engineered code was to set the number of mantissa bits of the real numbers to the native values of 23 bits for 32-bit reals and 52 bits for 64-bit reals. The program was built and run. Reassuringly, the simulated rocket stll rose to 237.446 feet. The emulation of the real arithmetic by calling subroutines and functions does not affect the result when the precision is not changed.

All of the working real variables in the program are 32-bit reals. The precision of the 32-bit reals was degraded, one bit at a time, and the program was run with each precision. The simulated maximum altitude is shown in Table 1.

Table 1. **met_rocket**: Altitude v Precision.

Precision (bits)	**Altitude (ft)**
Native (23 bits)	237,446
23	237,446
22	236,364
21	236,511
20	235,082
19	268,230
18	273,067

Degrading the precision from the native 23 bits to 22 bits causes a small (Less than 0.5%) change in performance. Degrading it to 19 or fewer bits causes significant change (At 19 bits the height changes by 13%). It seems that the 23-bit precision of the native 32-bit real numbers is only just sufficient for this simulation program. As a check, the program was converted to run entirely with 64-bit real numbers, with a mantissa of 52 bits. The simulated rocket rose to 237,786 feet. It seems that the (probably accidental) choice of 32-bit real numbers was just good enough.

This result is surprising. The author would have expected a simulation of this type to require 64-bit real numbers. But if 64-bit numbers had been used, and this were a larger, multi-processor simulation, then half of the inter-processor traffic would have been unnecessary.

4.2. A Second Example - ss_9b.f90, A Solar System Simulation

The program ss_9b.f90 is a nine body simulation of the solar system. The bodies are the Sun and the 8 major planets (Pluto is too small and remote to have much influence on anything else). It models the movements of the bodies and the gravitational forces between them. Again, this is a real program written for a real scientific project. The complete code is available at [19]. An extract from the declaration code is shown below:

```
REAL(KIND=kr),PARAMETER :: au = 149597870700.0D0
REAL(KIND=kr),PARAMETER :: gu = -6.67300D-11
REAL(KIND=kr),PARAMETER :: g = &
 & ((60.0D0*60.0D0*24.0D0)**2/au**3) * gu
REAL(KIND=kr)          &
  time                 &
, p_time               & ! Previous time
, dt                   &
, end_time             &
, mass(9)              &
```

The **ss_9b** program was re-engineered to emulate the real arithmetic. The corresponding section of the automatically re-engineered code is shown below:

```
TYPE(em_real_k8),PARAMETER :: au = &
 em_real_k8(149597870700.0D0)
TYPE(em_real_k8),PARAMETER :: gu = &
 em_real_k8(-6.67300D-11)
TYPE(em_real_k8),PARAMETER :: g = &
```

```
  em_real_k8(((60.0D0*60.0D0*24.0D0)** &
  2/au%value**3)*gu%value)
 TYPE(em_real_k8)       &
  time                  &
  ,p_time               & ! Previous time
  ,dt                   &
  ,end_time             &
  ,mass(9)              &
```

As before, the only change to the executable code is the insertion of a call to the subroutine which initialises the emulated arithmetic.

The original program, and the re-engineered code with a precision of 52 bits for 63-bit IEEE numbers, were both run in a simulation of the orbits for 10 years. The positions of the Sun and of the 8 planets were identical in the two runs. Again, the re-engineering and emulation of the real numbers does not affect the results.

The re-engineered code was then run, progressively reducing the number of bits of precision of the 64-bit real numbers. The radial positions of the first four planets in the plane of the ecliptic are shown in Table 2 below (Radii in Astronomical Units (AU), 1 AU is the mean radius of Earth's orbit, angles in degrees).

Table 2. `ss_9b`: Positions of the Inner Planets after 10 Earth Years.

mantissa bits	**Mercury**		**Venus**		**Earth**		**Mars**	
	Radius	Angle	Radius	Angle	Radius	Angle	Radius	Angle
52	0.4594	-103.57	0.7219	53.47	0.9914	-8.78	1.5899	-75.97
24	0.4594	-103.71	0.7219	53.43	0.9914	-8.78	1.5900	-75.97
23	0.4594	-103.72	0.7219	53.42	0.9914	-8.77	1.5896	-76.01
22	0.4593	-103.81	0.7219	53.59	0.9913	-8.81	1.5900	-75.99
21	0.4593	-103.99	0.7219	52.90	0.9914	-8.87	1.5897	-75.95
20	0.4594	-103.70	0.7221	52.40	0.9913	-9.04	1.5898	-75.99
19	0.4584	-105.49	0.7221	53.70	0.9914	-8.65	1.5898	-75.99
18	0.4561	-108.57	0.7217	55.25	0.9913	-8.77	1.5898	-75.98
17	0.4583	-104.20	0.7213	50.80	0.9912	-9.64	1.5858	-76.55
16	0.4128	87.52	0.7196	49.73	0.9902	-9.46	1.5932	-74.91

Almost nothing changes as the 52-bit precision of the 64-bit numbers is reduced to 23 bits! With only 19 bits of precision the simulation is just beginning to break down. Only when the precision is reduced to 16 bits are the planets nowhere near where they should be.

This result was completely unexpected. Again, if this were a multi-processor simulation, half of the inter-processor traffic would be wasted.

5. Finding Statements And Variables Where Precision Is Inadequate

5.1. The Approach - Tracing Program Execution

The objective is to identify those statements where reduced precision degrades the results beyond a criterion value. The procedure by which this is done is described by Collins et al [2013] [20]. The code is instrumented by inserting calls to sub-programs which capture every left-hand-side value of an intrinsic numeric type immediately after it is computed. The values are captured to file. An example of the instrumented code of `met_rocket.f` is shown below:

```
!          Pressure in atmospheres
           atm_pressure = 10.0D0**(-h/50850.0D0)
           CALL trace_r4_data(atm_pressure,31)
!
           vel = SQRT(xd*xd+hd*hd)
           CALL trace_r4_data(vel,32)
!          Drag linear with pressure, quadratic with velocity
           drag = atm_pressure* &
            (c_drag(2,stage)*vel**2+c_drag(1,stage)*vel)
           CALL trace_r4_data(drag,33)
!
           xdd = ((thrust-drag)/mass)*COS(theta)
           CALL trace_r4_data(xdd,34)
           hdd = ((thrust-drag)/mass)*SIN(theta)-g
           CALL trace_r4_data(hdd,35)
```

Routines such as **`trace_r4_data`** capture a variable to a trace file. The second argument of each routine is a unique identifier which enables the statement to be identified, and which allows the trace mechanism to check that the program flow follows the same path in different runs. Note that there is no need to capture the values assigned to aggregate derived types because the components of the types must be assigned before the variables of the derived types are assigned, but it is necessary to capture entire arrays when they are assigned in elemental expressions.

The program is run with native precision, and the trace file is captured. This is the reference trace. A test run is then made with modified precision. If the values computed are compared with those in the reference trace they will differ for two reasons: because the precision of the values has been degraded, and because of numerical drift.

Numerical drift occurs because small differences in values caused by the reduction in precision accumulate statistically as the program runs. The values computed slowly drift away from those in the reference trace, and the differences will eventually exceed any criterion difference. The effects of numerical drift must be eliminated in order to identify those statements where the precision of the results has been seriously degraded. They are elimated as follows.

The routines such as **`trace_r4_data`**, which captured the reference trace values in the first run, *read* the reference trace file in the test runs. The values computed are immediately compared with those in the reference trace file. Then:

1) If the values are the same, no action is taken;
2) If the values differ by more than a criterion amount a report is generated;

3) If the values differ at all, the value computed is overwritten by the value in the reference trace file.

Because the results are overwritten, the inputs to every statement are the same in the reference run and in the test run. Any large difference is due to a breakdown in computing the output values due to the reduced precision.

Two criteria are used to compare the values computed with those of the reference trace. If the reference trace value is below a threshold criterion, no report is made. This prevents the report of large percentage differences when numbers are close to zero. If the trace file value is above the threshold, the percentage difference between the numbers is computed and compared with a criterion percentage. If the criterion is exceeded then the value computed, the reference trace value and the unique identifier of the trace statement are all reported.

This technique was first used to eliminate numerical drift from program runs to expose coding errors and compiler bugs (Collins et al [2013] [20]. The first major trial was on WRF, the climate and weather modelling code maintained at The University Corporation for Atmospheric Research, Boulder, Colorado [2]. Note that WRF is a very large program. This technology is scalable.

5.2. Finding the Statements Where Computation Breaks Down

The results of met_rocket break down when the precision is reduced to 19 bits. The test for degraded computation was made, with a precision of 19 bits, progressively reducing the error criterion until failures were reported. The first failures reported are from a group of statements with the trace labels 32 to 44. The simulation runs for 120,340 time steps and each of these statements is executed once in each time step. Table 3, below, shows the number of failures reported for each of these statements in a simulation run with two error criteria. A failure is reported if the result differs from that from the full precision run by more than the criterion percentage.

Table 3. `met_rocket`: Points of Failure

Code position	**Number of failres with error criterion:**		
	20%	**10%**	**5%**
32	0	1	1
33	1	1	1
34	1	1	2
37	1	1	1
39	0	1	1
41	0	2	2
43	1	1	1
44	1	2	2

The statements where a 20% criterion is exceeded are those marked *20% below:

```
          vel = SQRT(xd*xd+hd*hd)
          CALL trace_r4_data(vel,32)
!         Drag linear with pressure and quadratic with velocity
          drag = atm_pressure* &
           (c_drag(2,stage)*vel**2+c_drag(1,stage)*vel)
*20%      CALL trace_r4_data(drag,33)
!
          xdd = ((thrust-drag)/mass)*COS(theta)
```

```
*20%       CALL trace_r4_data(xdd,34)
           hdd = ((thrust-drag)/mass)*SIN(theta)-g
           CALL trace_r4_data(hdd,35)
!
!          Trapezoidal integration - AB2
           xd = xd+frametime*(2*xdd-pxdd)
           CALL trace_r4_data(xd,36)
           hd = hd+frametime*(2*hdd-phdd)
*20%       CALL trace_r4_data(hd,37)
           pxdd = xdd
           CALL trace_r4_data(pxdd,38)
           phdd = hdd
           CALL trace_r4_data(phdd,39)
           x = x+frametime*(2*xd-pxd)
           CALL trace_r4_data(x,40)
           h = h+frametime*(2*hd-phd)
           CALL trace_r4_data(h,41)
           pxd = xd
           CALL trace_r4_data(pxd,42)
           phd = hd
           CALL trace_r4_data(phd,43)
           IF (vel .GT. p_launch_vel) THEN
              theta = ATAN2(hd,xd)
*20%          CALL trace_r4_data(theta,44)
           ENDIF
```

The largest errors occur in the atmospheric drag, **`drag`**, the down-range acceleration, **`xdd`**, the vertical velocity, **`hd`** and the attitude, **`theta`**.

The solar system simulation, **`ss_9b.f90`**, starts to break down when the precision is reduced to about 18 bits. The test for degraded computation, with the error ratio criterion set to 10%, shows 13 errors at statement index 13. The code is:

```
        DO body1 = 1,9
           CALL trace_i4_data(body1,10)
           DO body2 = body1+1,9
              CALL trace_i4_data(body2,11)
              DO axis = 1,3
                 CALL trace_i4_data(axis,12)
!                Difference in position
                 ds(axis,body1,body2) = &
                  s(axis,body1)-s(axis,body2)
*10%             CALL trace_r8_data(ds(axis,body1,body2),13)
!                Square of difference in position
                 s2(axis) = &
                  ds(axis,body1,body2)*ds(axis,body1,body2)
                 CALL trace_r8_data(s2(axis),14)
              ENDDO
```

The error in ds, the difference in position between two bodies along one axis, has exceeded 10%. This appears to be the cause of breakdown of the solar system simulation with decreased precision.

6. Additive Underflow

6.1. The Additive Underflow Mechanism

Additive underflow is a major cause of computational breakdown with reduced precision. It occurs in the addition or subtraction of real and complex values.

An IEEE real number is made up of 3 components. A sign bit, an exponent and a mantissa. When two real numbers are added or subtracted, the mantissa of the smaller number is shifted to the right by the number of bits required to make the exponents equal. If the magnitude of one number is very much larger than the other, many of the bits in the smaller number are shifted to the right of the mantissa bits in the result, and are therefore lost in the calculation.

6.2. Finding Statements with Additive Underflow

Additive underflow is detected in the Fortran functions used to overload addition and subtraction of real and complex objects. The code of the check for addition and subtraction of two 32-bit emulated reals is shown below.

```
ELEMENTAL SUBROUTINE check_underflow_r4_r4(r,a,b)
        IMPLICIT NONE
        REAL(KIND = kr4),INTENT(IN) :: r
        REAL(KIND = kr4),INTENT(IN) :: a
        REAL(KIND = kr4),INTENT(IN) :: b
        REAL(KIND = kr8) :: ta,tb,tmax,tmin,tr,td,te
        ta = ABS(a)
        tb = ABS(b)
        IF(ta>tb)THEN
           tmax = ta
           tmin = tb
        ELSE
           tmax = tb
           tmin = ta
        ENDIF
        tr = ABS(r)
        td = ABS(tr-tmax)
        IF(tmin>check_underflow_delta)THEN
           te = ABS(1.0D0-td/tmin)
           IF(te>check_underflow_crit)THEN
              CALL signal_underflow(te,tr,tmax,tmin)
           ENDIF
        ENDIF
END SUBROUTINE check_underflow_r4_r4
```

The value `te` is the proportion of the smaller term which is lost in the addition or subtraction. The routines which overload addition and subtraction may be called several times in processing a statement. Therefore, the detection of additive underflow is recorded internally when the check is made. It is reported after the statement in the routines which write the results to the trace file.

6.3. Additive Underflow in `met_rocket` and ss_9b

In **`met_rocket`**, the errors due to additive underflow actually become significant when the precision is reduced to 19 bits. With the error criterion set to 20% (i.e. where 20% of the change due to the smaller term is lost) the error counts are shown in Table 4 below:

Table 4. 20% Additive Underflow in **`met_rocket`**

Code position	Underflow count	Expression where underflow occurs
32	33	**`vel = SQRT(xd*xd+hd*hd)`**
34	6	**`xdd = ((thrust-drag)/mass)*COS(theta)`**
35	587	**`hdd = ((thrust-drag)/mass)*SIN(theta)-g`**
36	17277	**`xd = xd+frametime*(2*xdd-pxdd)`**
41	13949	**`h = h+frametime*(2*hd-phd)`**
45	1	**`time = time+frametime`**

The underflow at position 32, the velocity computation, occurs at the maximum height, where the vertical velocity is close to zero.

The underflow at positions 34 and 35, the thrust-drag computation, occurs at the launch, where the velocity is still very small so the drag is very small.

The underflow at 36, the down-range (Horizontal) velocity occurs as the vehicle approaches its maximum height where the atmospheric drag is very small.

The underflow at 41 occurs as the maximum height is approached. The change in height is small compared to the height itself so most of the mantissa bits of the change in height are lost. This is the dominant point of failure in the performance with reduced precision.

The solar system simulation, **`ss_9b`**, starts to break down when the precision is reduced to about 18 bits. With 18 mantissa bits and the underflow error criterion set to 20% (i.e. 20% loss of precision of the smaller term) all of the underflow errors are reported at one statement:

Table 5. 20% Additive Underflow in **`ss_9b`**

Code position	Underflow count	Expression where underflow occurs
15	4389	**`r2(body1,body2) = s2(1)+s2(2)+s2(3)`**

`r2` is the square of the radial distance between two bodies. **`s2(3)`** is the distance of a body from the plane of the ecliptic, which small compared with the distance from the centre of the solar system and, for the Earth, is usually very small. The result is unsurprising. Only when the underflow error criterion is reduced to 2% is underflow detected in another statement. The report shows:

Table 5. 2% Additive Underflow in **`ss_9b`**

Code position	Underflow count	Expression where underflow occurs

13	4275	**`ds(axis,body1,body2) = &`**
		`s(axis,body1)- s(axis,body2)`

15	80981	**`r2(body1,body2) = s2(1)+s2(2)+s2(3)`**

`ds` is the difference in position between two bodies along one axis, and underflow is expected to occur when one of the bodies passes close to a Cartesian axis. Again, the result is unsurprising. It appears that additive underflow may not be a significant cause of computational degredation with reduced precision in this program.

7. Execution Speed

The instrumented code runs very much more slowly than the original code, particularly when run-time trace data are captured to disk. The intention is to use this technique in experiments to identify the precision required for a program, and then to modify the production code to transfer less data between processors, and, if appropriate, to carry out computations at a reduced precision.

8. Conclusions, Implications And Further Work

8.1. Conclusions

It is possible **`to prove`** that a given precision, measured in the number of mantissa bits, is sufficient for execution of a given program to a given accuracy.

It is possible to identify the first statements and variables implicated in the breakdown of performance when precision is reduced. The promotion of the precision of specific variables may then free the choice of precisions for less critical variables.

8.2. Implications

Almost all commercially available computer systems use the IEEE 754 number formats. The implication of this study, and of the studies by Düben et al [2015] [7] and Dawson and Düben [2016] [8] are that lower precision number formats with shorter word lengths would be of considerable value. These would provide improvements in computational speed through faster arithmetic, and through improved packing of data in cache and in inter-processor communication. There is a strong case for the development of a lower precision real number kind for standard processors.

8.3. Further Work

The procedures described may be applied to a range of Fortran codes to verify that the numerical precisions chosen are adequate. It is also possible to identify sub-sets of variables where the precision may safely be reduced to improve execution speed. The authors of the engineering tool used in this study have agreed to make it available for academic work.

A simple application of the results is to modify the inter-processor communication routines, in particular mpi interfaces, to pack data at reduced precision. There will be a trade-off between the time taken to pack and unpack the data, and the time saved in data transfer. In multi-processor atmospheric and finite element programs it is often the case that each processor holds the state variables which describe a region of a system. Each processor transfers the values at the edges or its region to the processors which hold the data for neighbouring regions, where the values are used only for for derivative calculations. Reducing the precision of the values transferred in these cases is similar to the scale separation strategy used by the AD10 [15].

These procedures may also be used to support experiments with different number formats, for example, with integer arithmetic or scaled fractions.

REFERENCES

[1] IEEE Computer Society (1985) “IEEE Standard for Binary Floating-Point Arithmetic”, doi: 10.1109/IEEESTD.1985.82928, http://ieeexplore.ieee.org/servlet/opac?punumber=2355.

[2] The National Center for Atmospheric Research (2009) “WRF, Weather Research and Forecasting", web pages: http://www.wrf-model.org.

[3] John Collins, Brian Farrimond and Mark Anderson (2011) “Automatic Detection of Softare Errors in WRF”, poster presented at the 12th WRF Users Workshop, http://www.simconglobal.com/WRF_Workshop_June_2011_Poster_Automatic_Detection_of_Software_Errors_in_WRF.pdf

[4] John Collins, Brian Farrimond, Mark Anderson and David Gill (2012) “Q.A. Analysis of the WRF Program”, poster presented at the 13th WRF Users Workshop, http://simconglobal.com/WRF_Workshop_June_2012_Poster_QA_Analysis_of_the_WRF_Program.pdf

[5] John Collins, Mark Anderson, Brian Farrimond, Darryl Bayliss and Darryl Owens (2013) "Systematic Errors in Climate Models Consequent on the Design of the Fortran Language", Poster presented at the 4th WGNE Workshop on Systematic Errors in Weather and Climate Models, http://simconglobal.com/4th_WGNE_Workshop_April_2013_Poster_Fortran_Language.pdf

[6] Peter D Düben and T N Palmer (2014) “Benchmark Tests for Numerical Weather Forecasts on Inexact Hardware", Monthly Weather Review, Vol 142, No 10, pp 3809-3829. doi: 10.1175/MWR-D-14-00110.1.

[7] Peter D Düben, Francis P Russel, Xinyu Niu, Wayne Luk and T N Palmer (2015) "On the use of programmable hardware and reduced numerical precision in earth-system modeling", Journal of Advances in Modeling Earth Systems, Vol 7 No 3, pp 1393-1408. doi: 10.1002/2015MS000494.

[8] Andrew Dawson and Peter D Düben (2016) "rpe v5: An emulator for reduced floating-point precision in large numerical simulations", Geoscientific Model Development Discussions, November 2016, doi: 10.5194/gmd-2016-247.

[9] J Shukla, T N Palmer , R Hagedorn, B Hoskins, J Kinter, J Marotzke, M Miller and J Slingo (2010) "Toward a New Generation of World Climate Research and Computing Facilities", Bulletin of the American Meteorological Society, Vol 91, doi: 10.1175/2010BAMS2900.1.

[10] H S Cheng, W Wulff, A N Mallen, S V Lekach, A Stritar and R J Cerbone (1984) "BWR Plant Analyzer Development at BNR", BNL-NUREG—35843, http://www.iaea.org/inis/collection/NCLCollectionStore/_Public/16/064/16064663.pdf

[11] Ken Peterson (1984) "The Use of DAREA, The Data Area Optimizer, on a Jet Engine Simulation", Proceedings of the Applied Dynamics International User Society, http://www.simconglobal.com/peterson_1984.pdf

[12] Gottfried Hogh, Barry K Powell and Gerald P Lawson (1988) "Real Time Engine Dynamic Analysis and Control", SAE Technical Paper, 885104, doi: 10.4271/885104.

[13] Richard H Hanson (1984) "Raytheon's Implementation of a Missile Simulation using the AD10", Proceedings of the Applied Dynamics International User Society, http://www.simconglobal.com/hanson_1984.pdf

[14] Leslie Johnson (1987) “Missile Simulation with Actual Hardware Using AD10s", Proceedings of the Applied Dynamics International User Society. http://www.simconglobal.com/johnson_1987.pdf

[15] Edward O Gilbert et al. (1980) “AD10 Hardware Reference Manual”, Published: Applied Dynamics Internalional Inc. 3800 Stone School Road, Ann Arbor, MI, USA.

[16] Peter D Düben, Hugh McNamara and T N Palmer (2014) "The use of imprecise processing to improve accuracy in weather and climate prediction", Journal of Computational Physics. Vol 271, No C, pp 2-18, doi: 10.1016/j.jcp.2013.10.042.

[17] John Collins and Brian Farrimond (1989-2017) "fpt Reference Manual", SimCon Ltd. web pages: http://simconglobal.com/fpt_ref_index.html

[18] Stephen D Mandy (2006) “met_rocket,f example program”, web pages: http://simconglobal.com/downloads/met_rocket.f.

[19] John Collins (2013) “ss_9b.f90 example program”, web pages: http://simconglobal.com/downloads/ss_9b.f90

[20] John Collins, Brian Farrimond, David Flower, Mark Anderson and David Gill (2013) “The Removal of Numerical Drift from Scientific Models", International Journal of Software Engineering & Applications, Vol 4, No 2, doi: 10.5121/ijsea.2013.4204.

AUTHOR:

John Collins is a research associate at the Radar Remote Sensing Group at the University of Cape Town, an honorary research fellow at Edge Hill University and technical director of SimCon Ltd. UK. His primary interests are in compiler design and the analysis of computer programs.