Software Design and Development

Computer Arithmetic and Numerics

N.M. Maclaren

nmm1@cam.ac.uk

October 2018

3.1 Introduction

3.1.1 Stratospheric Overview

The numbers considered here fall into three main categories: the integers (whole numbers), as introduced in primary school; the real numbers, as used in secondary school science; and the complex numbers. If you are not familiar with complex numbers, you need not worry about them, as this course spends very little time on them, though you will probably have difficulty with other courses on this MPhil! It does assume that you are familiar with integer and real arithmetic, as used in all science degrees.

> *For pure mathematicians only*: the categories are the ring of integers of characteristic zero, $\mathbb{Z}$, and the complete fields of real and complex numbers, $\mathbb{R}$ and $\mathbb{C}$. No other categories of number are covered.

Computer hardware has limited approximations to these; for example, the integers are limited in size, and real numbers are represented by limited precision floating-point ones. The software (compilers, libraries etc.) often extends the hardware in many ways, such as by providing unbounded integers and building complex numbers out of real ones. The principles described here are largely language-independent, and apply to almost every computer language and application, including C, C++, Fortran, Matlab, Mathematica, Maple, Python, Perl, Java, Excel etc.

In mathematics, these categories of number are defined by (and define) their properties, such as associativity — i.e. (A+B)+C and A+(B+C) are equivalent, and so are (A*B)*C and A*(B*C). Almost all of the problems with computer arithmetic arise because the computer's approximation does not preserve all of the expected mathematical properties, and the programmer accidentally assumes something that is not true. This is a problem with software facilities as much as hardware ones, and with integers as much as with floating-point numbers.

3.1.2 DON'T PANIC†

Anyone put off by the pure mathematics terminology should not panic when it is used, as everything that is introduced will be explained in very simple terms. The explanation of the terms may help when reading books or Web documents.

† *With apologies to the late Douglas Adams.*

This may be a very complicated area, but this course is intended to be a map through the minefield. If you follow the recommendations, and take even moderate care, you can avoid most problems and recognise the ones that you cannot avoid. Recognising when you are getting into dangerous areas is the key to reliable programs and debugging, because there is no problem so hard to locate as one you do not even know can happen. Some problems you can do nothing about except be aware that they exist, and to check whether they have caught you out when things do go wrong. And, as we all know, Murphy's Law says that they will.

This lecture was derived from a longer course, which has a bit more detail and covers some less common topics; there is also a relevant Computer Science course. Both have some further reading:

How Computers Handle Numbers - a.k.a. Computer Arithmetic Uncovered
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/Arithmetic/`

`https://www.cl.cam.ac.uk/teaching/1718/NumMethods/nummeths17slides-asprinted.pdf`

As usual, contact your supervisor for advice, and I am happy for your supervisor to contact me.

3.1.3 Consistency/Sanity Checking

As mentioned in the previous lecture, one of the best defences against numeric problems is to put in a lot of consistency and sanity checking. This should check not just whether values are valid but also whether they are realistic; one of the first symptoms of numeric failure is often that intermediate values start being physically unrealistic.

There is no need to check everything, everywhere, because the aim is to detect failures early (i.e. fairly close to where they started) and not to identify the actual cause. It is enough to make debugging simpler, not to eliminate it ((a much harder task), so most tests can be fairly simple. Even the simplest tests often detect a failure cascade before it has destroyed all evidence of where it started. The diagnostics should include at least an identification of which test failed, so that you know where to start debugging. For example:

```
if (speed < 0.0 .or. speed > 3.0e8)    &
    call panic('Speed error in my\_function')
```

Do not be afraid to write your own classes, which need not use any more memory; modern compilers will compile them efficiently, by inlining functions. You can then check the values systematically. This is especially useful for arithmetics like complex, where you could check just multiplication and slower actions.

3.2 Integer Arithmetic

3.2.1 Basic Information

Paradoxically, problems with simple integer arithmetic cause as much trouble as those with the more complicated real and complex arithmetic, because most people do not expect

trouble. It is mostly trivial, and works just as you would expect. By far the two most important causes of problems are overflow and mixing signed and unsigned integers – the latter is a problem only for languages that have both concepts (mostly C-like), of course.

The only other area that causes significant problems is division and remainder, where at least one of the values is negative. The exact effects are language-dependent, so do check up before relying on them, or add a check and fix up the result. The value is always truncated towards zero if both values are positive. Division by zero is an error, of course, and so is taking the remainder with respect to zero.

Nowadays, every computer uses binary, twos' complement, integers. Binary means that numbers are expressed in base 2: e.g. using conventional ordering (i.e. most significant bit first) on 8 bits, $01010011 = 2^6 + 2^4 + 2^1 + 2^0 = 83$ (decimal). Twos' complement describes how negative numbers are implemented, and says that the most significant bit is treated as negative, so that $11000101 = -2^7 + 2^6 + 2^2 + 2^0 = -59$. Clean numeric code does not assume any of this, but many languages do expose the integer representation to programs, as described later.

Similarly, all hardware supports both 16- and 32-bit integers, which have ranges of -32768 to 32767 and -2147483648 to 2147483647, respectively), and most support 64-bit ones (c. -9.2×10^{18} to c. 9.2×10^{18}). A very few support 8-bit or 128-bit ones (-128 to 127 and c. -1.7×10^{38} to c. 1.7×10^{38}), but many compilers will emulate 8-bit integers by operating on them in 16 bits, and storing back only the least significant 8 bits. Some will emulate 128-bit integers using 64-bit ones.

Nowadays, every computer will wrap the result on overflow and continue execution (e.g. 2*83 = -90 and 4*83 = 76, quietly), but it usually sets a flag which can (in theory but rarely in practice) be tested by the compiler. Also, optimising compilers often rearrange code in ways that are mathematically but not computationally equivalent. The result is that an operation in a program's **code** may not wrap even when the **hardware** does; this is described in more detail later.

Options for trapping errors are described later.

3.2.2 Problems with Wrapping

Programmers using languages that wrap, quietly, on overflow need to know what it implies. For example, M > N may not mean M*M > N*N, and many other invariants may fail. Even if your code does not assume that, optimisation means that the compiler may, which has the consequence that overflow can make a program behave incorrectly in ways that does not look plausible when reading the code. Few people believe this, even when it happens to them.

A realistic example (i.e. I have seen it catch several people, including me) is when declaring a multidimensional array and calculating its total length. On a system which can address more memory than the size of an integer (e.g. a 64-bit system with 32-bit integers, which is a **very** common combination nowadays), the array may be definable but the size may wrap. This is a major trap in Fortran, C, C++ and probably other languages. For example, consider code like:

```
parameter (n = 1800)
double precision d(n,n,n)
call init(d,n*n*n)
```

If `n*n*n` wraps in 32 bits, this is equivalent to calling `init(d,1537032704)`, which means that only part of the array would be initialised, with consequential chaos. You can test for overflow fairly easily in several ways, such as checking that you get the same result in double precision real arithmetic, or dividing the total by a factor and checking the result. For example, in this case:

```
ntotal = n*n*n
if (ntotal /= n*dble(n)*n) call panic(...)
if ((ntotal/n)/n /= n) call panic(...)
```

3.2.3 Handling of Overflow

The biggest problem with computer integers is overflow, and at least 90% of all bugs associated with ordinary (signed) integer arithmetic are caused by the program mishandling it. Some languages avoid the problem (e.g. Python), and users of those languages need worry only about things like array overflow (e.g. attempting to access element 12345 of a 100-element array), not integer overflow as such. Options to trap overflow are described later.

In other languages (e.g. C, C++ and Fortran), integer overflow is **undefined**, and this is a major cause of wrong answers, crashes etc. It is important to realise that undefined means just that, and it does not mean that the values will wrap modulo $|2^{bits}|$, even when the hardware does. Simple tests for what happens are **usually** misleading, **most** books and Web pages are misleading, and undefined behaviour is **NOT** equivalent to system dependence. This point cannot be stressed too strongly.

The reason is that optimisation may cause anything to happen, and this includes when the compiler is running in its least optimised, debugging mode. Compilers do not simply map code to hardware operations, but produce sequences that have an equivalent effect, and they assume the mathematical properties of integers when doing that. Almost all language standards say that they can generate any code that is mathematically equivalent, but need do so only for valid operations, and it is the programmer's fault (and problem) if the result overflows. What is more, the effects are likely to change with compiler versions, as well as between systems.

This is not the course to go into details, but a compiler might well recognise code including the following and make the indicated assumptions:

- If D = (A*B)/C, then D/A = B/C
- If A > 0 and B > 0, then A*B > 0

But neither is true if A*B overflows, so the code will assume something that is not true, and chaos ensues. As mentioned above, few people ever believe this, even when faced with the evidence.

Let us give an over-simplified example. We have code like:

```
A = 5000 ;      C = 50000 ;
X = 2*A*C                   = 500000000          Right
Y = (A-C)**2 - A**2         = 2000000000         Right
...  = X/B+Y/B              = 20325202           Right
```

This fairly often actually compiles into code that executes like:

```
A = 5000 ;      C = 50000 ;
X = 2*A*C                   = 500000000          Right
Y = (A-C)**2 - A**2         = 2000000000         Right
...  = C**2/B               = 2865986            Wrong!
```

Printing out the value of the variables in a debugger does not help here; you have to inspect the actual assembler generated and look at the values in the registers.

3.2.4 Integer Bases (or Radices)

Integers can be written (whether for input or output) in any base: binary 01010011 is decimal 83 is octal (base 8) 133 is hexadecimal (base 16) A3. Sometimes this is explicit, using more-or-less well-understood conventions, as 2r01010011 or 0xa3, but not always. C and C++ have a rather nasty convention where a leading zero indicates octal, in some contexts only, as in 0133. Unix sometimes uses unqualified octal, which means that you need to know the context to decide what value 136 (or, for that matter, 0136) represents. Generally, most users will use decimal for formatted I/O, but it is quite easy to use any other base. Incidentally, the terms *base* and *radix* are interchangeable in this context — some authors use one and some the other!

In binary, bitwise `AND`, `OR`, `NOT` etc. make sense, so integers can be used as arrays of flag bits. This is a little unclean, but most languages support it nowadays. You are strongly advised to do this only with positive numbers; each language's rules on how negative ones are handled are different, they may be undefined or defined to have bizarre effects.

Similarly, left and right shift by N places are multiplication and division by 2^N. However, it is important to note that shifting is safe **only** if all of the arguments and results are positive and the number of places to shift is **smaller** than the number of bits in an integer. For example, using 8-bit twos' complement, do not trust 2r11000101 = -59 shifted right two places to be 2r11110001 = -14 or 2r00110001 = 49; similarly, 1234 shifted right 200 places will probably **not** not be zero (4 is more likely, but it could be anything). This lunacy is a relic of 1950s electronic constraints.

3.2.5 Unsigned Arithmetic

C, C++, Java, Perl and perhaps other languages support **unsigned** integer arithmetic. This is exactly like twos' complement, except that the top bit is **not** negated, so negative integers cannot be represented and we now have (in 8 bits) $11000101 = 2^7 + 2^6 + 2^2 + 2^0 = 197$. Most of the basic arithmetic is performed modulo $|2^{bits}|$ (i.e. overflow wraps modulo $|2^{bits}|$). Because there are no negative numbers, bitwise logical operations and shifts are defined for all values, but still use only shifts smaller than the number of bits in a word.

For mathematicians: this is **not** *GF(2^N)), nor is it true modular arithmetic, despite the occasional claim that it is one or the other by people who ought to know better.*

Because numbers are always non-negative, there are some strange effects, such as subtracting 5 from 3 gives a large positive number. Division and remainder (A modulo B) are not performed modulo $|2^{bits}|$ (*that statement is not likely to make much sense to non-mathematicians*), which has some subtle consequences.

More importantly, the interactions between unsigned integer types and both signed integer types and floating-point types are foul. A few details are given later, especially for C and C++, which also has the trap that the basic `char` type may be either signed or unsigned. There is more on this in the full course mentioned above. Most people should merely note that this area is a minefield and keep signed numbers (both integer and floating-point) and unsigned integers as separate as possible.

3.3 Floating-Point

3.3.1 Basics of Floating-Point

It may help if we go through the basic principles of floating-point. When used for display purposes, this is often called scientific notation, and the form we shall consider here uses a leading zero before the decimal point (i.e. not the most common scientific format). Numbers are of the following form:

$$sign \times mantissa \times base^{exponent}$$

The mantissa is treated as a fixed-point number strictly less than 1 and greater than or equal to the inverse of the base (i.e. the first digit after the decimal point is non-zero). Such numbers are called *normalised*; if the first digit after the decimal point is zero, they are called *unnormalised* or *denormalised*. It is then multiplied by the sign and scaled by the exponent to give the value; for example, $+0.12345 \times 10^2 = 12.345$. Zero is represented specially. Here, we shall consider only formats that use a fixed precision, P (the number of digits after the decimal point); these may be called the significant digits, which is a little confusing. Such numbers have a relative accuracy of $\times(1 \pm base^{1-P})$.

A easier way of thinking of this is to regard the sign and mantissa as a fixed-point number between -1 and 1, exclusive (i.e. neither -1 nor 1 is possible), scaled by the exponent. This ignores the precision, but can be useful when designing programs, as this simpler model is generally enough to analyse the overflow behaviour.

3.3.2 Floating-Point versus Reals

Floating-point is not mathematical real arithmetic and the differences are important. The easiest (and usually the best) way to think of floating-point arithmetic is that it is effectively non-deterministic to within the accuracy limits — i.e. the results are guaranteed to be within those limits, but are otherwise unpredictable — which means that relying on the precise value is almost always a mistake. However, all of the differences lead to either a result very close to the expected one (i.e. within $\times(1 \pm base^{1-P})$), or arise only for numbers so near zero that they cannot be represented properly.

The above statement was uncontroversial 30 years back, but is now regarded as heresy; the current dogma is to confuse consistency and accuracy, so you will read precisely the converse in many or most modern books.

The reason that the difference is important is that fixed-point numbers break many of the rules of real arithmetic, and floating-point ones break even more, so ordinary mathematical notation cannot simply be mapped to program code. Wrong assumptions cause wrong answers, and a regrettably high proportion of published papers are based on results that are completely meaningless, for this reason alone. The key is to think in terms of floating-point rules, not the ones of real arithmetic — this is easier to do than it sounds, as 60 years of Fortran experience shows! Write what you mean, and do not assume that something that is mathematically equivalent is necessarily numerically equivalent; with a bit of practice, you will avoid most problems without having to think about them.

3.3.3 Invariants

Invariants are properties that always hold, and are very important and useful in mathematics and software engineering. Let us start with the properties that both floating-point and real numbers share.

- Both are commutative: i.e. A+B = B+A , and A*B = B*A.
- Both have zero and unity: i.e. A+0.0 = A, A*0.0 = 0.0, and A*1.0 = A.
- Both have negatives (additive inverses): i.e. each A has a B = –A, such that A+B = 0.0.
- Both are fully ordered: i.e. A $\geq$ B and B $\geq$ C means that A $\geq$ C, and A $\geq$ B is equivalent to B $<$ A.

Unfortunately, the differences are rather more extensive.

- Floating-point numbers are neither associative nor distributive: i.e. (A+B)+C may not be A+(B+C), (A*B)*C may not be A*(B*C), (A+B)–B may not be A, (A*B)/B may not be A, and A+A+A may not be 3.0*A.
- They are not continuous (for any of addition, subtraction, multiplication or division): e.g., for addition, B $>$ 0.0 may not mean A+B $>$ A, A $>$ B and C $>$ D may not mean A+C $>$ B+D, and A $>$ 0.0 may not mean 0.5*A $>$ 0.0.
- They have no multiplicative inverse: i.e. not all A have a B = 1.0/A, such that A*B = 1.0. And 1.0/A can fail for very small non-zero A.

Now, the above is true for all fixed-size floating-point, whether it is implemented on a computer or written by hand, but I will bet that most of it was not mentioned at school. Some other differences will be mentioned later, especially with regard to exceptional (error) values, which are very different.

Current Floating-Point Hardware

The current floating-point standard is called IEEE 754 (IEEE 854 is a minor variation), and is also known as ISO/IEC 10559, and almost all computers use it nowadays. See:

`http://754r.ucbtest.org/standards/754.pdf`

This is binary (i.e. the base or radix is 2) and signed magnitude (i.e. the sign bit is just that, and it is not twos' complement). The details are messy and so will be omitted here. It has 32- and 64-bit formats, with sometimes 80- or 128-bit extended formats (see later). The accuracies and ranges of the standard formats are:

32-bit, 4 byte, 'single precision': the accuracy is 1.2×10^{-7} (23 bits), and the range is 1.2×10^{-38} to 3.4×10^{38}

64-bit, 8 byte, 'double precision': the accuracy is 2.2×10^{-16} (52 bits), and the range is 2.2×10^{-308} to 1.8×10^{308}

The format includes representations of exceptional values, which indicate errors and other odd states, and they are discussed later.

The use of other sizes of floating-point is not recommended. There is no major problem in principle, but there are serious portability, accuracy and performance problems. Currently, IEEE 754 dominates people's thinking. You may come across 128-bit IEEE 754R floating-point (in several different variations), and it may be **very much** slower than 64-bit, because many of the operations may be emulated by trapping the instruction – or the mathematical functions may not be as accurate as they should be. You should also avoid native Intel native 80-bit floating-point, which is generally becoming less used, for good reasons. And there are plenty of others, which are even more obscure.

3.3.4 Very Small Numbers and Underflow

This is a complicated and messy area, but causes relatively few problems. The reason is that it generally does not matter what value is returned for a negligible result as long as it **is** negligible. It can cause performance problems and occasional program misbehaviour, but rarely causes wrong answers.

Denormalised numbers were introduced by IEEE 754, but are not always implemented on nominally "IEEE 754" systems. They are just numbers with the minimum possible exponent but zeroes after the decimal point — for example, in an artificial decimal representation, 0.00123×10^{-308}. Whether or not they are supported, numbers that are too small to represent in the chosen format are quietly replaced by zero (this is called underflow); underflow is never trapped nowadays, though it used to be, because so many codes fail if it is.

There are some numeric advantages to allowing denormalised numbers ("soft" underflow), and some disadvantages, but the claims that one approach is numerically better than the other are pure dogma. They can be very slow, because they are often not implemented in hardware, and the system takes an interrupt to fix them up in software, so there is often option to simply replace by zero ("hard" underflow). Few programs notice the difference.

A good rule is to use double precision to minimise the chances of falling into a trap — 10^{-308} is much less likely than 10^{-38} — and, if you do have trouble, it is almost always

safe to replace very small numbers by zero. Perhaps the worst trap is that most programs and applications compiled for hard underflow go wrong if presented with a denormalised number, so it is important not to import one into such programs when using binary I/O. And, of course, do not mix code compiled for hard and soft underflow in the same program.

Examples of the sort of thing that can cause trouble when numbers approach zero are:

- A small reduction in the size of a number may lose accuracy or turn it into zero. For example, (A/2.0)*2.0 may not be either exactly A or 0.0, and A > 0.0 does not mean
  2.0*A > 1.5*A > A.
- Two numbers may be different but their difference may be zero. For example, B > C does not mean B-C > 0.0.

3.3.5 Error Handling and Exceptions

Old maps are supposed to have marked uncharted waters where travellers had disappeared without trace with *Here Be Dragons* — this area needs the same annotation.

The following is what the ordinary user of floating-point **needs** to know; I wish that it were not so, and it used not to be the case, but it is now. Most of the details have been omitted but we will return to a few aspects later. There is more on this in the full course mentioned above.

IEEE 754 formats include both positive and negative zeroes, positive and negative infinities, and *NaN*s (Not-A-Number). Except as described below, it is possible to ignore the signs of zero, and users are **very** strongly recommended to do just that; at best, the sign of zero is grossly unreliable.

The infinities represent values that have overflowed. In simple cases, they imply numbers that are too large to represent, but this is not a reliable assumption. For example, `exp(1000.0)` will overflow and deliver an infinity, and taking the logarithm of plus infinity will also deliver plus infinity. But `log(exp(1000.0))` really represents the value 1000.0, which is not very large.

NaNs represent the result of an erroneous operation, typically because it is mathematically invalid. They are not as simple as that, but explaining their complexity is well beyond the scope of this course. Treat them as simple indications of an invalid calculation, representing a value that is numerical nonsense.

In theory, both propagate appropriately, and you can tell if a complicated calculation failed because the result is a NaN but, in practice, the error state is not reliable. The reasons for this are fiendishly complicated and tied up with the politics of international standards (IEEE 754, C99 and others), but the executive summary is that this area is a minefield. This course attempts to explain how to traverse it with only a moderate risk of getting your foot blown off.

3.3.6 What Can Be Done?

I shall give a partial solution to the problem before describing the problem in detail. The pedantic answer to *What Can Be Done?* about this situation is *Not A Lot*, because

the problem is caused by broken language standards and specifications. However, there are some programming techniques that can help reduce the damage.

The first is to put in consistency and sanity checking, as described above — because one bogus value breeds others, such checks often work very well. The second is to use double precision to reduce the likelihood of overflow — because 10^{308} is so much larger than 10^{38}, this often helps a great deal. Note that this can often speed up a program, by avoiding underflow and denormalised numbers, which often interfere with pipelining even when they do not cause interrupts.

But do not assume that the first time your checking catches a bogus value is the first time that one has appeared, or that a failure to catch any means that there were none. No matter how careful you are, some will always slip through the net, so always cross-check your results whenever possible. Note that these remarks apply to **all** categories of error, including array bound overflow and pointer problems.

3.3.7 Divide by Zero, Infinities etc.

At school, you were taught that dividing by zero is an error, but C, C++ and Fortran often do not treat it as one; in fact, C99 in 'IEEE' mode is specified not to, because IEEE 754 states that both overflow and divide-by-zero give an infinity. This makes sense when implementing interval arithmetic, but is a very serious trap for ordinary programmers. To implement this, zero is signed in IEEE 754 and the sign is regarded as meaningful, but this ignores the fact that those language standards permit a compiler to rearrange code and remove obviously irrelevant operations in a way that will reverse the sign. For example, if we have:

B = A–A; C = –B; D = C+0.0

then:

B = C = D = 0.0

but:

1.0/B $\neq$ 1.0/C and 1.0/C $\neq$ 1.0/D

Compilers will often optimise code like `1.0/(A+0.0)` to `1.0/A`; after all, they **are** mathematically equivalent. The result of this is that you should **never** trust the sign of an infinity, and it would be a much better specification if the languages returned a NaN when dividing by zero. But, because they do not, this is a trap for the unwary, and the sign of infinities is essentially meaningless.

If anyone doubts this, try the following Fortran program or its equivalent in your favourite language. The output suddenly switches from "–infinity" to "+infinity".

```
program fred
double precision ::  x = -1.0d-300
do k = 1,25
x = x-0.9d0*x
print *, 1.0d0/x
end do
end program fred
```

Because the root cause of this problem is that zero's sign is treated as meaningful, but it generally is **not** meaningful, exactly the same problem applies to functions that test sign bits, such as Fortran `SIGN` and C `copysign`. Those are even worse, because they will extract the sign from NaNs, and IEEE 754 specifies explicitly that the sign of a NaN is undefined.

3.3.8 Trapping

It is sometimes possible to trap such errors, as well as integer overflow. If you can, do so, because it can save a **lot** of wasted effort. In IEEE 754 terms, the serious errors are overflow, division by zero and invalid; the others should generally be ignored.

NAG Fortran always traps serious arithmetic errors.

With GNU `gcc/g++`, use `-ftrapv` to trap integer overflow, but be warned that it will catch only **some** of them, because of the signed/unsigned morass. Nowadays, `-fsanitize=undefined` is better, because it will trap many other errors as well, including the shift-related ones. If your compiler is new enough, you can add `,float-divide-by-zero,float-cast-overflow` after the `undefined`, but there is no way to trap floating-point overflow or invalid exceptions, unfortunately. With `gfortran`, use `-ftrapv -ffpe-trap=invalid,zero,overflow` to trap integer overflow and the serious floating-point errors.

With Intel use `icc/icpc -ffpe-trap=invalid,zero,overflow` or `ifort -fpe0` to trap the serious floating-point errors; it never traps integer overflow. With `numpy` use `seterr(over='raise',divide='raise',invalid='raise')` to trap the floating-point errors; you can use `call` rather than `raise` if you prefer. With `C#` use the `checked` keyword or option to trap some integer overflow.

3.3.9 NaNs and Error Handling

The general principle of IEEE 754 is that invalid operations should result in a NaN, and that operations on NaN arguments should return NaNs. For example, 0.0/0.0, infinity/infinity and infinity–infinity all return a NaN. With luck, taking the square root of a negative number will, too. This is numerically respectable, and corresponds to the behaviour of the 1960s and 1970s statistical and engineering applications that pioneered the use of error values. Unfortunately, it does not clearly separate its numeric primitives from its semi-numeric ones (e.g. those that are needed to operate on NaNs specially), and using the latter makes it very easy to lose the NaN state by accident.

C99 and, to a lesser extent, Java and other C-derived languages have taken the approach that, if a non-NaN result can possibly be justified, then it should be preferred. This means both that it does not return a NaN when it should (e.g. atan2(0.0,0.0) returns $-\pi$, -0.0, +0.0 or $+\pi$, depending on the signs of the zeroes) and provides a great many ways in which the NaN state is lost, quietly. See:

How Java's Floating-Point Hurts Everyone Everywhere
`http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`

Be careful when reading that. Professor Kahan may be the world's leading numerical computationalist, but many people believe that his designs (including IEEE 754) are sharp-edged tools for numerical experts only, and not safe tools for use by the inexperienced.

He is unquestionably correct that the lack of either traps or flags, compiled with IEEE 754 value semantics, makes Java numerically very dangerous — and the same is true of C99 (and increasingly C++). The result of all this is that you should not trust compilers and applications to detect their errors, and should not even trust them to return or display a NaN when they have performed an invalid operation. Here are a few examples of the many traps for the unwary:

- `int(`*NaN*`)` is often `0`
- `max(`*NaN*`,1.23)` may be `1.23`
- Comparisons on a *NaN* usually deliver false, quietly

I once tried writing some numerical code using the last convention, seeing if I could do it reliably. When I analysed it, I had got the NaN handling of 20% of the tests wrong. As I somewhat arrogantly regard myself as one of the best numerical coders around, I do not have any confidence that most people will do better. The comparison issue has another problem — take the first example of a sanity checking function:

```
if (speed < 0.0 .or. speed > 3.0e8) call panic(...)
```

That test will **not** work if `speed` is a NaN, because both comparisons will probably yield false. Now, in theory, we could change it to:

```
if (speed != speed .or. speed < 0.0 .or. speed > 3.0e8) call panic(...)
```

That is perhaps the cleanest solution, but many compilers will optimise that test out, so it is not reliable. Or we could test for validity and negate the result using `.not.`, but probably the safest approach is to write the ungainly code:

```
if (speed >= 0.0 .and. speed <= 3.0e8) then
    call evaluate(...)
else
    call panic(...)
endif
```

However, cautious users will not trust even that, especially with high levels of optimisation. I am afraid that there really is no good solution — despite claims, C99 does not help, and C++ is following it.

3.4 Miscellaneous Information

3.4.1 Complex Numbers

Generally, these are simple to use, but that does **not** apply to C99, for reasons that are too complicated to go into here. They are invariably implemented in software as (real,imaginary) pairs of floating-point numbers. Fortran and C++ have them built-in, and C99 more-or-less does.

I/O is usually done on the raw floating-point numbers (which is a bit unclean), it is easy to lose the imaginary part by accident, and special functions can be slow and unreliable, but those are relatively minor problems. The biggest one is that exception handling is uniformly untrustworthy — often catastrophically so. If you even approach the limits, the usual effect is that you will get mathematically wrong answers, quietly.

> *For pure mathematicians only*: the complex numbers are compactified by adding a single point at infinity, turning them into the Riemann sphere, but this conflicts with using the Cartesian representation.

An analysis of this is far too foul for this part of the course, but interested users should try to write a function to perform complex division that gets correct answers right up to the limits. It is not a pretty problem. But you will discover that it is easy to produce code that works very well for almost all values, and will go wrong only for ones that are almost overflowing (within a factor of two of it). If you can avoid producing such numbers, then you will have no problems. There is more on this in the full course mentioned above.

So my recommendation is not to trust the exception handling an inch, and to put sanity checks into your code, but otherwise not to worry.

3.4.2 Mixed Type Expressions

Converting signed integer to floating-point to complex is usually not a problem, provided that the precisions are adequate. When converting N-bit integers to N-bit floating-point, and the values are such that they cannot be held precisely (above 8.3×10^6 in 32-bit or 4.5×10^{15} in 64-bit), you may come across some extremely weird rounding. That is about the worst problem. Converting unsigned integers to floating-point usually works, but a few implementations are buggy when the top bit is set (i.e. when the number would be negative if it were signed).

Going the other way is trickier. Floating-point to integer truncates towards zero, in almost every modern language; that is the easy bit. Overflow is undefined in C, C++ and Fortran; the effect is completely unpredictable, and can include plausible but weirdly wrong integers. Converting complex to floating-point takes the real part in all of those languages, quietly, thus providing an opportunity to lose the imaginary part by accident (a **very** common error, especially in languages with implicit declaration).

Converting infinities and NaNs between any of integer, floating-point and complex is effectively undefined in C, C++ and Fortran, in any direction, and there is no such thing as typical behaviour. You do not want to know what C99 says about complex infinities and NaNs, but the executive summary is that they are more broken than you would believe possible. Fortran says nothing about them, which is a vast improvement. There is more on this in the full course mentioned above.

So the recommendation is the same as for exception handling in complex arithmetic, though with extra care when converting floating-point to integer.

3.4.3 Formatted Input/Output

Formatted output is generally safe and, in this context, output includes all conversions of numbers to text (including string types). The conversion accuracy of very large or small numbers is not always good, though this rarely causes trouble. Infinities and NaNs are usually handled correctly nowadays, but not always in the way you might expect.

The most common trap is that some people do not realise that values like 0.1 are not exact in binary – decimal 0.1 is binary 0.0001100110011001... (i.e. binary 0.0$\underline{0011}$ recurring). Because of the base mismatch, only 15 digits are guaranteed reliable for 64-bit floating-point (6 for 32-bit), but 18 (9 for 32-bit) digits must be printed if the number is to be read in again unchanged. So do not trust 10.0*0.1 to be exactly 1.0, and remember that how many digits you print depends on what you want to do with the output.

And, of course, you should check on infinities, NaNs and denormalised numbers before relying on them; if the implementation is poor, it will fail with those.

Formatted input (including all conversions from text to numbers) is far more of a problem than output, because it needs to deal with bad input. Format errors and overflow are often undefined, and the application or compiler will often fail to detect either, or handle them sanely even when it does. The behaviour can be very weird indeed.

Infinities, NaNs and denormalised numbers are always unreliable, and you should never trust the implementation without checking. It is good programming practice to include a minimal cross-check yourself on all data that you read in, to pick up typing mistakes, but it will also help pick up conversion errors. You are recommended to check both the values for sanity, and that the count of numbers is what it should be.

3.4.4 Binary (Unformatted) I/O

This just writes the bits in their internal format to file and reads them back again; it is fast, easy and preserves the value precisely. Because it uses the internal format, do not use it between systems without checking, as you will be caught out sooner or later. Unfortunately, it often depends on the compiler and its options or the application, so it is not entirely safe even within the same system (except when using a single program).

Different languages use different methods, and Fortran is very different from C and C++; solutions do exist to transfer between them, but you may need help. Derived and other complicated types may add extra problems; assuming that those do not arise, we can now give an almost complete checklist for how to use this safely.

The systems, applications or languages must use same formats, the same sizes and the same endianness. The formats are defined by the application or language and perhaps its options. The sizes are defined by those, and whether the system is 32- or 64-bit, whether it is Unix, a Microsoft Windows system or MacOS and, again, perhaps the system's options — for example, you can create 32-bit programs under most 64-bit systems.

Endianness is whether the bits of integers are stored least significant first ("little endian") or most significant first ("big endian"). Floating-point numbers are usually stored in the same order as integers, and complex numbers are always stored in the order real, imaginary. Common little endian systems include Intel x86 and clones (including AMD,

AMD64 and Intel EM64T) and Alpha; common big endian ones include Sun SPARC, SGI MIPS, HP PA-RISC and IBM PowerPC. Intel Itanium can be either, just to confuse the issue, and the mixed endian systems are probably all now defunct.

And, to reiterate, do check on the denormalised number, infinity, and NaN handling before relying on it, and do not import them into programs that are not expecting them, or anything may happen.

One of the best ways of doing this is to create a file with the 'numbers' 0.0, $\pm 10^k$ ($k = -323 \ldots +308$), $\pm$inf and NaN, read them in, and calculate all 8 million combinations of the basic operations, add, subtract, multiply, divide and compare of. You should get the same results on both systems! The crudest method is to print them out to (say) 12 digits in a fixed format and compare them using *diff*, but this can be done more intelligently.

3.4.5 Single Precision (32-bit)

Do **not** use single precision for serious calculations unless you have analysed your problem carefully. The first reason is that, even if you need less than 6 digits of precision, that cancellation, error accumulation and ill-conditioning mean that it is quite likely that your results may not even have their first digit correct. The second is that you are much more likely to trip across exceptions and, as described above, that is Bad News. Let us look at a trivial but realistic example for the accuracy point:

The quadratic $x^2 + 10^4 \times x + 1$ has roots of c. 10000 and 0.001; now if we apply the usual formula $(-b \pm \sqrt{b^2 - 4ac})/(2a)$ in 32-bit, we get roots of c. 10000 and zero. This may not matter, but it might lead to a division by zero, giving infinity and not c. 1000, which could then make subsequent results wildly wrong.

Another problem is that modern machines have lots of memory, and this allows for big problems; even the most stable of big problems need more accuracy. For example, the error in the solution of linear equations or eigensystems is often of the form:

$$precision \times matrix\_dimension \times condition\_number$$

In single precision, the first is 1.2×10^{-7}, the second could easily be 10^4 and even well-behaved matrixes may have a condition numbers of 10^3. Oops. Even Fourier transforms are susceptible to this problem.

On GPUs, single precision is a lot faster than double (sometimes 10 times or more), and you may **need** to use it for performance. Some problems are very stable (numerically), and you will have no problem – but, in general, this is a **major** headache. The first approach is always to check if there is a more stable algorithm; they can be difficult to find, and may be slower. If that does not work, there are precision-extension techniques that were commonly used 30 years ago and more, and now are needed again. These require some fairly extreme hacking, unfortunately, but I wrote some code which is available (for inspection or use) at:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/Development/Programs/double_emulation.cpp`

And, finally, consider a different mathematical approach, which is **not** a task to undertake lightly!

3.4.6 Numerical Issues

You can assume that your results will almost never be better than your input — the old acronym GIGO stands for *Garbage In, Garbage Out* and is entirely correct. But you should **not** assume that your result is accurate to even the minimum of your input accuracy and the machine precision, as errors can often build-up rapidly.

The NAG library is most general reliable library, by a very large margin; the Numerical Algorithms Group is the leading numerical software house in the world. There are some very good open-source libraries (e.g. LAPACK), but many others are seriously unreliable or worse. For reasons that I am not going into now, do **not** trust Numerical Recipes , most especially not the first edition. The same applies to the Web, which has not been called The Web of a Million Lies for no reason. In both cases, some parts are good, and some are ghastly.

Numerical problems arise everywhere non-trivial, including linear equations, determinants, eigensystems, solution of polynomials, regression, analysis of variance, ODEs, PDEs, finite elements etc. Any method will work in simple, small cases, but poor ones will fail in complex or larger ones; this does not make testing simple. There is a great deal more on this in the next lecture.

There are a few things that can be done which help.

- Put consistency checks into your program, such as cross-checking that the solution of an equation returns reasonable values for the parameters and that it really does satisfy the equation to within reasonable accuracy bounds! This also helps to speed up debugging.
- Use high-quality algorithms and libraries, as recommended above. Some of them will give error estimates, and it is often worth trying a slower but more accurate method if your answers seem weird. If the results change, you have a problem.
- Try perturbing your input — not just the data, but the parameters you have copied from the literature — and see if that changes the results unreasonably. If so, you have a possible problem. Unfortunately, if there is no change, it does **not** mean that there is no problem – see paragraph 5 in:

How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?
`http://www.cs.berkeley.edu/~wkahan/Mindless.pdf`

- As always, seek expert advice, which does **not** mean the most self-confident of your colleagues. Until you know which are reliable and which are unreliable, be careful. Ask your supervisors or Directors of Studies for whom to contact, when necessary.