How Computers Handle Numbers

A.k.a. Computer Arithmetic Uncovered

Nick Maclaren

nmm1@cam.ac.uk

May 2011

Stratospheric Overview

Integers ($\mathbb{Z}$), reals ($\mathbb{R}$) and complex ($\mathbb{C}$)
Hardware has limited approximations to them
Software extends hardware in many ways

Principles are largely language–independent
Apply to Python, Perl, Java, Excel, Matlab, C, . . .
. . . C++, Fortran, R, C#, Maple, Mathematica etc.

But mathematics and computing don't match
Not just floating–point, nor even just hardware

DON'T PANIC

Course will give a map through the minefield

With moderate care, can avoid most problems
Course helps to recognise dangerous areas

May help to debug when things do go wrong
Knowing that something may happen is key

- Some problems you can only watch out for

Will give guidelines on how to do that

Beyond the Course

Arithmetic/
Follow the link for further information /break
http://www.cl.cam.ac.uk/teaching/1011/FPComp/

There is some further reading in both of those
A few reasons are available – optional

Numerical Coding Book

Real Computing Made Real:
Preventing Errors in Scientific and
Engineering Calculations

by Foreman S. Acton

Good, clear book on avoiding precision loss etc.
Explains only how to prevent some forms of error!

Doesn't overlap with this course much

Consistency/Sanity Checking (1)

- Put in lots of this, kept simple

E.g. check values are valid and realistic

- Pref. every entry/exit of major code unit

Check most data being used/returned/changed

- No need to check everything, everywhere

Aim is to detect failures early and locally

```
if (speed < 0.0 .or. speed > 3.0e8) &
      call panic("Speed error in my_function")
```

Consistency/Sanity Checking (2)

Ideally, something like:

```
def prevaricate (delay, reason) :
    check_delay(delay)
    check_reason(reason)
    . . .
    excuse = . . .
    check_reason(excuse)
    return excuse
```

Consistency/Sanity Checking (3)

- Write sanity checker for major data structures

Easy to add checking calls for debugging

```
call sanity_upper (n, a, lda)
call sanity_rect (n, nrhs, b, ldb)
call dposv ('u', n, nrhs, a, lda, b, ldb, info)
call sanity_upper (n, a, lda)
call sanity_rect (n, nrhs, b, ldb)
```

$O(n^3)$ calculation – $O(n^2)$ checking cost

Benefits of Checking

May double time taken to get code to compile
AND halve total time until it mostly works!

- Not restricted to numerical aspects

An old "software engineering" technique
Predates that term by many decades . . .

Won't cover any more of this here, but see
Debugging/

Using Classes

Don’t be afraid to write your own class
You don’t need to use any more memory
Modern compilers will compile that efficiently

- You can then check the values systematically

Especially useful for arithmetics like complex
Could check just multiplication and slower actions

- Don’t forget to initialise on creation

Where Do Problems Arise?

Paradoxically, often for integer arithmetic!
People get careless with simple aspects

Real (i.e. floating–point) is a lot trickier
Most people are aware of that, in theory

- But it isn't as tricky as often thought

60 years of Fortran use shows that one!

Complex is a little trickier, but not much

Integers

- Mostly trivial, and just work as you expect
This course skips all of the simple aspects
Only three areas cause significant trouble

- Almost all problems arise with overflow

- Followed by signed/unsigned problems
This affects only some (C–like) languages

- Followed by the division/remainder rules

Will mention a few advanced features, as well

Division/Remainder Rules

If both M and N are positive, M/N rounds down
And (M/N)*N+remainder(M,N) = M

- Language–dependent if either are negative

Check its specification if you depend on that
Alternatively write a run–time test, and fix up

And, of course, division by zero is an error
Consequently, so is remainder by zero

That's all . . .

Unlimited Size Integers

- No limit on size, except memory and time

Built–in to Python, BigInt in Perl
Libraries (e.g. GMP) for C, C++, (Java, Fortran?)
Also Mathematica, Maple, bc etc.

Good packages are easy to use

- Eliminates overflow complexities
- But indefinite growth will crash program

And, only if you use very big numbers:
multiply/divide/remainder/conversion slow

Current Integer Hardware

Binary, twos' complement, e.g. for 8 bits:
01010011 = $2^6 + 2^4 + 2^1 + 2^0$ = 83
11000101 = $-2^7 + 2^6 + 2^2 + 2^0$ = −59
16, 32 and 64 bits, rarely 8 and 128 bits

Overflow wraps: 2*83 = −90 and 4*83 = 76

- Your CODE may not wrap – see later

⇒ Means that (M*N)/N may not be M
And other, similar, invariants may fail

Problems with Wrapping

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

Assume 64–bit system with 32–bit integers
Very common environment nowadays
Equivalent to calling init(d,1537032704) – Oops!

- Can't avoid, so must watch out for it – how?

Checking for Wrapping

Either of the following will detect it

- Both cost very little in effort or time

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

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

- Even checking for negative bounds helps

Will pick up over half of such cases!

Integer Overflow (1)

Some always use floating–point (Excel, Matlab)
May convert to floating–point (Perl, R)
Convert to unlimited size (Python but not numpy)
Very rarely, trap it and diagnose the failure

- All fairly safe options for most use

May wrap modulo $|2^{bits}|$ (Java, C#, numpy)

- Generally NOT what you want (see later)

May be UNDEFINED (C, C++, Fortran)

Integer Overflow (2)

Be warned: wrapping modulo $|2^{bits}|$ is dangerous
Any optimisation can cause truly horrible effects
Even with none, there are some very nasty gotchas

- Sometimes option to trap it, diagnose and stop

NAG Fortran always does, gfortran –ftrapv enables it
gcc –ftrapv and g++ –ftrapv will trap some overflows

- Using C# checked keyword raises an exception

- These are the best solutions, when available

Undefined Behaviour

Major cause of wrong answers, crashes etc.

- Effects are almost always unpredictable

Even unrelated differences may have effects

- Sometimes debuggers misbehave or crash
- Simple tests are usually misleading
- Most books / Web pages are misleading

Undefined behaviour $\neq$ system dependence

Reasons are beyond this course – please ask

Over-Simplified Example

B = C = D = 5000

A = B*C*D = 445948416 Wrong

E = A/D = 89189 Wrong

print E 89189 Wrong but consistent

Fairly often actually compiles vaguely like:

A = B*C*D = 445948416 Wrong

E = A/D = 89189 Wrong but consistent

print E ⇒ print B*C

print E 25000000 ⇐ But this is E!

Integer Formatted I/O

- Representation not usually important

Most people never need to know it

Can read or display in any base:
Bin. 01010011 = dec. 83 = oct. 133 = hex. A3
May be explicit: 2r01010011 or 0xa3

Most formatted I/O is done in decimal, anyway!

Unix may use octal – what is 136? Or 0136?

Using Integers as Bits

You can treat integers as arrays of bits

But not in Matlab or R, for good reasons

Bitwise AND, OR, NOT etc. make sense

Can even mix bitwise and arithmetic operations

All well–defined, portable and reliable

- Except for negative numbers

Keep all numbers non–negative and in–range

Negative numbers are for language lawyers

Shifting

Shift of N is multiply/divide by 2^N

- Don't shift negatives or through sign bit

It may work, but each language differs

- Keep all shifts below number of bits in word

Python is a rare exception to this

See the extra foils for why – it's bonkers!
A relic of 1950s electronic constraints

Unsigned Integers

Mainly for C, C++, (& Java, Perl) users

Arithmetic modulo 2^{bits} (not GF(2^N))
In 8 bits, 11000101 = $2^7 + 2^6 + 2^2 + 2^0$ = 197
As for hardware, numbers wrap round at 2^N

Numbers are always non–negative – e.g. 3–5 > 0

- Divide/remainder aren't modular

- Pure unsigned arithmetic is fairly safe

Mixing Signed and Unsigned

- Signed/unsigned interactions are foul
Conversions are usually not what you expect

- It's very tricky to avoid mixtures in C/C++
Another C/C++ warning – char may be either
More details for C/C++ in extra information

- A minefield in all languages that have it
C/C++ people need to watch out for 'gotchas'

Fixed-Point Arithmetic

Fixed number of digits after decimal point
Precision is part of variable's type
Usually implemented as scaled integers

Heavily used for financial calculations
Rare in scientific computing, but in bc/dc etc.

Generally easy to use, except for:

- Rounding of multiplication/division
- Mixing precisions, conversion, etc.
- Special functions (sqrt/log/etc.)

Scaled Fixed-Point

Fixed–point with a separate scale factor
Common in 1950s – replaced by true floating–point

C# decimal has resuscitated it
Possibly using IEEE 754 decimal floating–point

- Almost always, it's a complete waste of effort
True fixed–point or floating–point are better

It's closely related to unnormalised floating–point
Also a proposed DEC64 format (not covered further)

Rational Arithmetic

One of the main modes in Mathematica
Combined with unlimited size integers

Only serious problem is explosion of size
Otherwise, it works just as you would expect

Fixed size rationals have their advantages
Sometimes called fixed–slash arithmetic
Really esoteric – ask offline if interested

Basics of Floating-Point

Also called (leading zero) scientific notation

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

E.g. $+0.12345 \times 10^2$ = 12.345

1 > mantissa $\geq$ 1/base ("normalised")

P sig. digits $\Rightarrow$ relative acc. $\times(1 \pm base^{1-P})$

Also –maxexp < exponent < maxexp – roughly

Like fixed–point –1.0 < sign/mantissa < +1.0

Scaled by $base^{exponent}$ (10^2 in above example)

Floating-Point versus Reals

- Floating–point effectively not deterministic
Predictable only to representation accuracy
Differences are either trivial – $\times(1 \pm base^{1-P})$
Or only for infinitesimally small numbers

- Fixed–point breaks many rules of arithmetic
- Floating–point breaks even more
Wrong assumptions cause wrong answers

- The key is to think floating–point, not real
Practice makes this semi–automatic

Invariants (1)

- Both are commutative:

A+B = B+A, A∗B = B∗A

- Both have zero, unity and negation:

A+0.0 = A, A∗0.0 = 0.0, A∗1.0 = A

Each A has a B = −A, such that A+B = 0.0

- Both are fully ordered:

$A \geq B$ and $B \geq C$ means that $A \geq C$

$A \geq B$ is equivalent to NOT $B > A$

Invariants (2)

The following are approximately true
Don't assume that they are exactly true

- Neither associative nor distributive:

(A+B)+C may not be A+(B+C) (ditto for *)

(A+B)−B may not be A (ditto for * and /)

A+A+A may not be 3.0*A

Invariants (3)

- They do not have a multiplicative inverse:

Not all A have a B = 1.0/A, such that A*B = 1.0

- Not continuous (for any of +, −, * or /):

B > 0.0 may not mean A+B > A

A > B and C > D may not mean A+C > B+D

A > 0.0 may not mean A > 0.5*A > 0.0

Remember School Maths?

Above is true for all fixed–size floating–point
Whether on a computer or by hand in decimal

- But were you taught that at school?

It doesn't cause too much trouble
But it does take some getting used to

Current Floating-Point Hardware

IEEE 754 a.k.a. IEEE 854 a.k.a. ISO/IEC 10559
http://754r.ucbtest.org/standards/754.pdf
Binary, signed magnitude – details are messy

- 32–bit = 4 byte = single precision

Accuracy is $\mathbf{1.2 \times 10^{-7}}$ (23 bits),
Range is $\mathbf{1.2 \times 10^{-38}}$ to $\mathbf{3.4 \times 10^{38}}$

- 64–bit = 8 byte = double precision

Accuracy is $\mathbf{2.2 \times 10^{-16}}$ (52 bits),
Range is $\mathbf{2.2 \times 10^{-308}}$ to $\mathbf{1.8 \times 10^{308}}$

Other Sizes of Floating-Point

- Don't go there – ask if you might need to

IEEE 754 dominates people's thinking

May have 128–bit IEEE 754R floating–point
In several different variations . . .
It may be very much slower than 64–bit

Exact FP arithmetic usually futile (explosion)
Interval arithmetic trendy but little better
Arbitrary precision is easy, but out of fashion
but Mathematica has it (almost unusably)

Intel/AMD Arithmetic

- Avoid it completely if you can

Generally becoming less used

Compilers/packages often use it internally

- One cause of differences in results

80–bit: accuracy is 1.1×10^{-19} (63 bits),

Range is 3.4×10^{-4932} to 1.2×10^{4932}

Typically stored in 12 or 16 bytes (96 or 128 bits)

http://www.intel.com/design/...

.../pentium4/manuals/index_new.htm

Decimal Floating-Point (1)

Added to IEEE 754R at IBM's instigation
Both IBM and Intel were going to put it in hardware
One Python module emulates it (in software)
It is beginning to look doubtful that it will take off

- It is NOT a panacea – OR any worse

Exactness claims (Python etc.) are propaganda
Try π, 1.0/3.0, $\mathbf{1.01^{25}}$, scientific code

It is claimed to help emulate decimal fixed–point

- That is complete and utter hogwash

Scientific programmers aren't interested, anyway

Decimal Floating-Point (2)

In binary floating–point, if $a \leq b$:

$a \leq a/2 + b/2 \leq b$ & $a \leq (a+b)/2 \leq b$

But not necessarily in decimal floating–point

The other “gotchas” are extremely arcane

It may look more accurate, but it isn’t

Writing portable code is easier than it appears

NAG was base–independent before 1990

But Intel have dropped it and IBM has backed off

- Will it ever be relevant to scientists? Probably not

Denormalised Numbers

- Only in IEEE 754 systems, and not always
Minimum exponent and zeroes after point
E.g., in decimal, 0.00123×10^{-308}
- Regard numbers like that as mere noise

- Replaced by zero if too small (underflow)
Never trapped nowadays – codes fail if it is

- Numeric advantages and disadvantages
Can be very slow – may take interrupt
Often option to always replace by zero

Denorms and Underflow

- Not generally a major problem
Use double precision to minimise traps
Almost always safe to replace by zero

(A/2.0)*2.0 may not be A

A > 0.0 does not mean 2.0*A > 1.5*A

B > C does not mean B–C > 0.0

And many others . . .

- Hard underflow code mishandles denorms
See later about binary I/O

Error Handling and Exceptions

Here be dragons ...

The following is what you NEED to know
Most of the details have been omitted
Will return to a few aspects later

- PLEASE contact me if you hit a problem

Other Exceptional Values

Zeroes are signed – but try to ignore that

- ±infinity represents value that overflowed

Not necessarily large – e.g. log(exp(1000.0))

- NaN (Not–a–Number) represents an error

Typically mathematically invalid calculation

In theory, both propagate appropriately

- In practice, the values are not reliable

What Can Be Done?

Consistency/sanity checking – yes, Yes, YES!

- Double precision reduces overflow problems

Can run faster, by avoiding exceptions/denorms

- Don't assume first catch is first failure
- Don't assume no catches means no failures

The above rules apply to most classes of error
E.g. array bound overflow, pointer problems

Floating-Point Overflow

Mathematica uses a fancy format and rarely overflows
Excel delivers "NUM!"
NAG Fortran always traps overflow
Some other compilers have a trapping option

All others deliver an infinity of right sign
numpy default gives a warning but not an exception

In itself, that would be perfectly reasonable and safe
    I.e. it's just using the affine extension of the reals
$\Rightarrow$ But remember the optimisation problems!

Divide by Zero etc.

Python, Perl, Excel, Matlab, Mathematica trap A/0.0
C, C++, Fortran rarely do (except for NAG)
Java, R, C# don't treat it as an error!

⇒ If not, divide–by–zero also gives infinity

numpy behaves exactly as for overflow

The sign of the infinity depends on the sign of zero
This is claimed to be "meaningful" – ha, ha!

Infinities and Errors

If we have B = A−A; C = −B; D = C+0.0;
All of B = C = D = 0.0
But $1.0/B \neq 1.0/C$ and $1.0/C \neq 1.0/D$

- ⇒ Don't trust the sign of infinities

- If you can, trap errors, diagnose and stop

In IEEE 754 terms, the serious errors are:
Overflow, divide by zero and invalid

Trapping

NAG Fortran always traps arithmetic errors

With gfortran/gcc/g++, use

–ffpe–trap=invalid,zero,overflow

With Intel (ifort/icc/icpc), use –fpe0

With numpy use:

seterr(over='raise',divide='raise',invalid='raise')

Or can use 'call' rather than 'raise'

Advanced Example

```
program fred
    double precision :: x = −1.0d−300
    do k = 1,6
        x = x−0.9d0*x
        print *, 1.0d0/x
    end do
end program fred
```

```
−1.0E+301
−1.0E+302
. . .
−1.0E+308
−Infinity
. . .
−Infinity
+Infinity
+Infinity
```

Signs of Zero and NaN

The same applies to functions that test signs

- Functions like Fortran SIGN, C copysign

And many others in C99 and followers

The signs of zeros and NaNs are interpreted
Never mind that those signs are meaningless

- Regard the result as an unpredictable value

See the extra information for more details

NaNs and Error Handling

Invalid operations may result in a NaN
0.0/0.0 = infinity/infinity = infinity−infinity = NaN
Operations on NaNs usually return NaNs

- But NaN state is very easy to lose

C99, Java actually REQUIRE it to be lost

Few examples of MANY traps for the unwary
int(NaN) is often 0, quietly
max(NaN,1.23) is often 1.23
Comparisons on NaNs usually deliver false

Sanity Checking and NaNs

if x != x then we have a NaN – in theory
In practice, may get optimised out

But don't make all tests positive checks
First example in course would be better as:

```
if (speed > 0.0 .and. speed < 3.0e8) then
      continue
else
      call panic('Speed error in my_function')
endif
```

Complex Numbers

- Generally simple to use (but C99's aren't)

Always (real,imaginary) pairs of FP ones
Python, Fortran, C++, Matlab, R, C99 (sort of)
Optional package for Perl, Java
Fortran usually most efficient for them

I/O usually done on raw FP numbers

- Easy to lose imaginary part by accident

Special functions can be slow and unreliable

- Don't trust exception handling an inch

It will often give wrong answers, quietly
Reasons are fundamental and mathematical

Mixed Type Expressions

Integer ⇒ float ⇒ complex usually OK
N–bit integer ⇒ N–bit float may round weirdly

Float ⇒ integer truncates towards zero
Complex ⇒ float is real part

- You won't generally get any warning

Overflow is undefined in C , C++ , Fortran
Java is defined, but very dangerous
Other languages are somewhat better

- Infinities and NaNs are Bad News

Complex and Infinities or NaNs

- This is a disaster area, to put it mildly

Don't mix complex with infinities or NaNs
All such code is effectively undefined

- That means float ⇒ complex, too

If the former has any of the exceptional values

See the extra information for some sordid reasons

- Regard complex overflow as pure poison

Put in your own checks to stop it occurring

Other Arithmetics

Let's use Hamiltonian Quaternions as an example

- Not going to cover them in this course!

Very few languages have them built–in
Can get add–on packages for most languages
Type extension can make look like built–in types

- Almost no extra problems over complex numbers

Main difference is that are not commutative

Other advanced arithmetics are similar
For example, true Galois fields and so on

Formatted Output

Generally safe (including number ⇒ string)

- Accuracy of very large/small may be poor

- Values like 0.1 are not exact in binary

Decimal 0.1 = binary 0.0001100110011001...
Only 6/15 sig. figs guaranteed correct
But need 9/18 sig. figs for guaranteed re–input

- Check on infinities, NaNs, denorms

If implementation is poor, will fail with those

Formatted Input

Far more of a problem than output

- Overflow and errors often undefined

Often doesn't detect either or handle sanely
Behaviour can be very weird indeed

Infinities, NaNs, denorms are always unreliable
Don't trust the implementation without checking

- Always do a minimal cross–check yourself

Undefined Behaviour and I/O

Generally, I/O conversion is predictable

- But only for one version of one compiler

But does mean that you can rely on tests

Actual conversion is in library, not code

All sharing compilers may behave the same way

Any upgrade may change behaviour

- It's worth preserving and rerunning tests

Binary (Unformatted) I/O

Shoves internal format to file and back again
Fast, easy and preserves value precisely

- Don't use between systems without testing
- Depends on compiler, options, application

Different languages use different methods
Solutions exist for Fortran ⇔ C
Derived/fancy types may add extra problems

- Can give almost complete checklist

Checklist for Binary I/O

- Must use same sizes, formats, endianness

Sizes are 32/64–bit mode, precision etc.

Formats are primarily application or language
Basic data types use the hardware formats
Derived types depend on the compiler etc.

"Little endian": Intel/AMD, Alpha
"Big endian": SPARC, MIPS, PA–RISC, PowerPC
Either: Itanium Mixed: dead?
May be compiler/application conversion options

Cross-Application Issues

Most compilers & applications are compatible
Cross–system transfer can be tricky
All systems now use very similar conventions

- But there are occasional exceptions

Especially with Fortran unformatted I/O

You probably won't hit problems with this
If you do, ask for help – it's not a big problem

IEEE 754 Issues

May be problems with denorms, infinities, NaNs
Can be chaos if code can't handle them

- Easy to write a simple test program

Just write an unformatted file with them in
Read it in, and check that they seem to work

0.0, $\pm 10^k$ ($k = -323 \ldots +308$), ±inf, NaN
Compare, add, subtract, multiply and divide
on all pairs – c. 8 million combinations
Crudely, print 12 digs, and use diff

Single Precision (32-bit)

- Do NOT use this for serious calculations

Cancellation / error accumulation / conditioning
Much more likely to trip across exceptions

$x^2 + 10^4 \times x + 1$ roots are c. 10000 and 0.001
$(-b \pm \sqrt{b^2 - 4ac})/(2a)$ in 32–bit
Delivers c. 10000 and true zero – oops!

- Lots of memory allows for big problems

Even stable big problems need more accuracy
1.2×10^{-7} often multiplied by matrix dimension

GPU Issues

Single precision is a lot faster than double

- You may need to use it for performance

- Some problems are very stable – no problem

But, in general, this is a major headache

- First check for a more stable algorithm

- There are precision–extension techniques

Commonly used 30+ years ago, now needed again
Ask your supervisor to contact me if it might help

Numerical Analysis (1)

Analyses effects of approximate calculations
Not covered here – DAMTP has 3 courses on it

Recommended to use a package or library:

- NAG library is most general reliable library
- Good open–source libraries (e.g. LAPACK)
- Many others are seriously unreliable or worse
- Do NOT trust Numerical Recipes or the Web

Arithmetic/

Numerical Analysis (2)

Many good, often old, numerical analysis books
Many are hard going and expensive or out–of–print
Following is good, affordable and available

Numerical Methods That Work: an Introduction
to Numerical Techniques and Problems
by Foreman S. Acton

Problem is it really IS an introduction
And, even then, it's not exactly bedtime reading!

Accuracy and Instability

Results almost never better than input (GIGO)

- Do NOT assume machine precision in result

Errors can often build–up exponentially

Single $\Rightarrow$ double may not help

- In that case, must improve algorithm

Trivial (not very realistic) example:

K'th differences of x^K, x=0.5,0.51,...,1.99,2.0

In D.P., 1 sig. fig. at K=7, nonsense thereafter

Cancellation (1)

- Low–level cause of most loss of accuracy

Caused by subtracting two nearly–equal values

Obviously, includes adding two with different signs

- But also dividing (and multiplying by inverse)

Assume numbers have P digits of precision

X and Y have Q leading digits in common

$\Rightarrow$ X–Y and X/Y–1.0 have precision P–Q

- Restructuring expressions can help a lot

Cancellation (2)

Where it matters, consider changes like the following:

(X+D)**2–X**2 ⇒ (2*X+D)*D

x^5–y^5 ⇒ (x^4+x^3*y+x^2*y^2+x*y^3+y^4)*(x–y)

sin(x+d)–sin(x) ⇒ sin(x)*(cos(d)–1.0)+cos(x)*sin(d)

I haven't used this, but you might like to try:
http://herbie.uwplse.org/

Cancellation (3)

- Watch out for large summations, too

Look up Kahan summation for a better method
I use another, or emulate extended precision:
C++/...
.../Exercises/Chapter_24/fancy_accumulate.cpp
.../Exercises/Chapter_24/fancy_inner.cpp

Unfortunately, it may be implicit in the algorithm
Common with ones that use numerical derivatives

Realistic Cases of Problem

Linear equations, determinants, eigensystems
Solution of polynomials, regression, anova
ODEs, PDEs, finite elements etc.

- Any method works in simple, small cases

Poor ones fail in complex, larger ones

- Put consistency checks in your program
- Use high–quality algorithms and libraries
- Try perturbing your input and check effects
- As always, find out what the experts advise

Topics Not Covered (1)

The details of any of the above topics
Too many other topics to list

Examples of areas that could have courses:

Parameterisation in C, C++, Fortran etc.
Interval arithmetic and its uses
Introduction to numerical analysis
C99 and its consequences

Topics Not Covered (2)

Older or rarer systems/problems/issues
Number handling in external protocols
Model, use and analysis of IEEE 754
IEEE 754R and decimal floating–point
Interactions with operating systems
Implementation techniques and implications
Mathematical models of computer arithmetic
And so on . . .

Reminder – Trapping Options

NAG Fortran traps everything by default

For gfortran/gcc/g++ use
–ftrapv –ffpe–trap=invalid,zero,overflow

For Intel (ifort/icc/icpc), use –fpe0

For Python numpy use
seterr(over='raise',divide='raise',invalid='raise')

For C# use checked keyword or option