Floating-Point Computer arithmetic

(Numerical Analysis MA228)

January 16, 2007

Main temes:

Defininition of the floating-point numbers (decimal, binary, arbitrary base), IEEE 754 standard, single and double precision, special numbers (zero, infinity, NAN), arithmetic operations, roundoff error and ways to minimise it.

Floating point numbers

Floating point (FP) numbers is a *finite* set of rational numbers designed to approximate real numbers in a computer. The easiest way to understand FP numbers is by the way of analogy with the scientific notation of decimal numbers. In the scientific notation, decimal numbers are represented as

$$Number = sign \times normalised\_mantissa \times 10^{integer\_power}.$$

The natural normalisation convention is when $1 \leq normalised\_mantissa < 10$, i.e. we require that the first decimal digit on the left is not zero. For example

$$-387.23510098 = -3.8723510098 \times 10^2.$$

The advantage of the scientific notation is that the *relative* error of approximating a real number is bounded and determined by the number of digits retained in mantissa, whereas in the fixed-point representation the number of retained digits bound the *absolute* but not the relative error. Basically, the same reason motivates using the FP representation of real numbers in computers.

In analogy with the decimal scientific notation, we introduce a general base-β FP representation as follows,

$$Number = s \times man \times \beta^{exp},$$

where $s = \pm 1$ is the sign, $1 \leq man < \beta$ is the normalised mantissa and exp is an integer exponent. We already saw that the scientific notation corresponds to base $\beta = 10$. In principle, there may be different choices of β in particular arithmetic implementations. However, by far most common choice in computers is binary, $\beta = 2$. This choice is used in the IEEE standard described in the next section.

IEEE binary Floating-point arithmetic

In most computers (with notable exeption of IBM mainframe and CRAY supercomputers) FP numbers are represented in a hardware conforming to IEEE[1] 754[2]. Particularly, it is followed by many CPU and FPU implementations (typically on all PCs). It was developed around 1985, with the aim to help floating-point computations be as reliable and correct as possible, and also to permit efficient (in terms of both machine time and also analyst time).

- sign bit (0 for "+" and 1 for "-").
- exponent: biased
- mantissa (without the most significant bit)

Binary floating-point numbers are stored in a computer as a string of bits arranged into a computer *word* as follows,

Sign bit	e exponent bits	m mantissa bits
1	1000 0101	11011010100000000000000

(The particular sequence of bits given in this example is a binary single precision FP representation of decimal number - 118.625).

The most significant bit is the sign bit, followed by the biased exponent, followed by the mantissa without its most significant bit. Storing the most significant bit in the mantissa would be redundant, see below).

Exponent biasing

The exponent is biased by $2^{e-1} - 1$. Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge

[1]IEEE = The Institute of Electrical and Electronics Engineers or IEEE (pronounced as eye-triple-e) is an international non-profit, professional organization for the advancement of technology related to electricity.

[2]The full title of the standard is IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985), and it is also known as IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems (originally the reference number was IEC 559:1989).

values, but two's complement, the usual representation for signed values, would make comparison harder. To solve this the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison. For example, to represent a number which has exponent of 17, the biased exponent is $17 + 2^{e-1} - 1$.

Cases

The most significant bit of the mantissa is determined by the value of exponent. If $0 < exponent < 2^{e-1}$ (i.e. not all 0's or all 1's), the most significant bit of the mantissa is 1 , and the number is said to be *normalized*. If exponent is 0, the most significant bit of the mantissa is 0 and the number is said to be *de-normalized*. Three special values arise:

- 1. if exponent is 0 and mantissa is 0, the number is ± 0 (depending on the sign bit)
- 2. if $exponent = 2^{e-1}$ and mantissa is 0, the number is $\pm\infty$ (again depending on the sign bit), and
- 3. if $exponent = 2^{e-1}$ and mantissa is not 0, the number being represented is not a number (NaN).

In the example given above we have a normalised number. Its mantissa is obtaned by adding 1 as the most significant bit on the left, i.e.

$1.11011010100000000000000 = 1\times2^{0}+1\times2^{-1}+1\times2^{-2}+0\times2^{-3}++1\times2^{-4}...+0\times2^{-23}$.

Features

- **different precisions:** float (i.e. single), double, extended, extended double, long double, ...
- **rounding modes:** default = to nearest ties to even, to zero, to $+\infty$, to $-\infty$; hardware implementation helps efficient tracking of round-off errors through changing round-off mode, or implementing interval arithmetics;
- **exceptions:** overflow, underflow, division by zero, illegal ops leading to NaN (e.g. square root of a negative number, arcsin of a number greater than 1, any op invoving NaN as one of the operands). In contrast with

past computing, the program is not aborted when exceptions occur. Now default behaviour is to continue with the computations. [3];

- **special values resulting from exceptions:**
  - ± 0 result from underflow, $\pm\infty$ result from overflow. Note that the sign is retained even though all significant bits are lost. However in comparisons, it is considered that $+0 = -0$, thus breaking $x = y \Leftrightarrow 1/x = 1/y$. Another special value is NaN which arises from an illegal operation. Note however that division by zero leads to infinity (with a respective sign) rather than NaN. Also, dividing a finite number by -0 leads to $-\infty$ etc.
- **exact rounding** of +,-,*,/,SQRT: the value returned by these functions is required to be the *exact* result of the operation correctly rounded according to the chosen rounding mode, eg.

$$x \otimes y = round(xy)$$

  However, for transcedental functions, exact rounding may not be required, due to the so called "table makers dilemma": there is no general mathematical procedure to know the result of $round(\ln x)$ for example.

Single precision case.

In this case FP has 32 bits with 8 bits allocated for the exponent and 23 bits for the mantissa (24 including the hidden most significant bit). The exponent bias now is 127 (in other words the exponent is stored with 127 added to it).

The smallest non-zero positive and largest non-zero negative numbers (represented by the denormalized value with all 0s in the Exp field and the binary value 1 in the mantissa field) are

$$\pm 2^{-149} \approx \pm 1.4012985 \times 10^{-45}.$$

The smallest non-zero positive and largest non-zero negative *normalized* numbers (represented with the binary value 1 in the Exp field and 0 in the mantissa field) are

$$\pm 2^{-126} \approx \pm 1.175494351 \times 10^{-38}.$$

[3] compare this with the Ada language aborting on exception: On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed about 40 seconds after takeoff. Media reports indicated that the amount lost was half a billion dollars – uninsured.

The largest finite positive and smallest finite negative numbers (represented by the value with 254 in the Exp field and all 1s in the mantissa field) are

$$\pm(2^{128} - 2^{104}) \approx \pm 3.4028235 \times 10^{38}.$$

Here is the summary table from the previous section with some example 32-bit single-precision examples:

Type	Exponent	Mantissa	Value
Zero	0000 0000	000 0000 0000 0000 0000 0000	0.0
One	0111 1111	000 0000 0000 0000 0000 0000	1.0
Denormalized #	0000 0000	100 0000 0000 0000 0000 0000	5.9×10^{-39}
Largest normalized #	1111 1110	111 1111 1111 1111 1111 1111	3.4×10^{38}
Smallest normalized #	0000 0001	000 0000 0000 0000 0000 0000	1.18×10^{-38}
Infinity	1111 1111	000 0000 0000 0000 0000 0000	∞
NaN	1111 1111	010 0000 0000 0000 0000 0000	NaN

Double precision case.

In this case FP has 64 bits with 11 bits allocated for the exponent and 52 bits for the mantissa (53 including the hidden most significant bit). The exponent bias now is 1023.

The smallest non-zero positive and largest non-zero negative numbers (represented by the denormalized value with all 0s in the Exp field and the binary value 1 in the mantissa field) are

$$\pm 2^{-1074} \approx \pm 5 \times 10^{-324}.$$

The smallest non-zero positive and largest non-zero negative *normalized* numbers (represented with the binary value 1 in the Exp field and 0 in the mantissa field) are

$$\pm 2^{-1022} \approx \pm 2.2250738585072020 \times 10^{-308}.$$

The largest finite positive and smallest finite negative numbers (represented by the value with 2046 in the Exp field and all 1s in the mantissa field) are

$$\pm(2^{1024} - 2^{971}) \approx \pm 1.7976931348623157 \times 10^{308}.$$

Some IEEE 754 features are difficult to exploit

IEEE 754 standard offers an impressive range of new features (only partly mentioned above) but some of them are difficult to exploit. Namely, until recently, most programming languages (or programming environments) didn't

have adequate support for permitting the user to control and take advantage of the hardware features present. Today, the C99 and Fortran 2003 standards go a long way in the right direction.

Typical problems were (are):

- no explit control of precision of variables and intermediate results, since floating-point registers of the CPU often accomodate exetended precisions while memory variable don't; this may lead to double rounding which can be undesirable;
- no way to set rounding mode
- no mechanism to trap floating-point exceptions (except a flag).
- register spills
- over-zealous optimization by the compiler: using algebraic identities which don't hold for rounded arithmetic, spilling wide registers to narrow storage.

You can check for yourself how your favorite programming environment behaves, be it C, C++, Fortran, Java, Matlab, and whatever tools (compiler, etc.) you usually use.

Exercise 1 (Can you specify the rounding mode ?) *Can you specify explicitly the rounding mode for floating-point operations ? Can you do this for each individual operation in the program ?*

Exercise 2 (Intermediate destination ambiguity) *Can you exhibit a situation when two different results are produced depending on if you store an intermediate result in a variable or not, due to different rounding precision (typically extended for intermediate results in a floating-point register, and not extended for variables) ? (Hint: the simplest is to iterate division by 2; in the simplest loop you typically get extended precision, but if you make it more complicated by introducing dummy variables, at some point you get non extended precision; not all platforms have double precision, so this may not always occur).*

Exercise 3 (Over-zealous optimization) *Give an example of over-zealous optimization when the computer is using algebraic identities valid for real but invalid for the FP numbers (e.g. associativity of arithmetic ops).*

Round-off

The weird properties of machine arithmetic: usual algebraic identites are broken. In general (ie. it happens for a significant fraction of typical values of x, y, z, where these designate machine numbers):

- addition and multiplication are not associative

$$(x \oplus y) \oplus z \neq x \oplus (y \oplus z) \;\; and \;\; (x \otimes y) \otimes z \neq x \otimes (y \otimes z)$$

- we lose distributivity of addition with respect to multiplication

$$x \otimes (y \oplus z) \neq (x \otimes y) \oplus (x \otimes z)$$

- dividing then multiplying by the same number does not return the original result

$$(x \oslash y) \otimes y \neq x$$

- converting from decimal to binary and then back to decimal does not return the original number; same for binary to decimal to binary conversion.

Exercise 4 (Floating-point breaks arithmetic identities) *Give examples to all of the above (without overflow, underflow, division by zero or invalid operation), in binary, supposing precision of the mantissa is 5 bits, and minimal and maximal exponents are respectively* $e_{\min} = -14$, $e_{\max} = 15$ *(ie. a 5 bit exponent with -15 bias, but these details are not important here).*

Some useful definitions:

- **ulp:** Unit in the Last Place: for a given machine number $x = (s, exp, man) = s \times man \times 2^{exp}$, this is the quantity (in absolute value) by which x changes if we change its last digit by 1, that is 1ulp $= 2^{-m+exp}$. Ulp can be seen as the maximum possible *absolute* error between a real x and its machine representation $fl(x)$.

- **machine epsilon:** ϵ: largest *relative* error between a real x and its machine representation $fl(x)$, supposing that the latter is a normal (in particular its mantissa is normalized, ie. has a single non-zero digit before the radix point) machine number. This is also the largest relative error corresponding to 1/2 ulp. Thus

$$\epsilon = \max(\frac{1}{2}2^{-m}/man) = 2^{-1-m}/\min(man) = 2^{-1-m}$$

- **wobble:** is the quotient of largest and smallesr relative error. For a general β-base FP number the *wobble* is

$$wobble = \beta$$

This means that the bigger the base β, the less uniformly numbers are spaced. This makes error analysis slightly more difficult for larger β.

Round-off errors and tricks to get rid of them

Round-off errors propagate and get amplified after multiple application of the arithmetic operations. Kahan algorithm of summation offers a clever way of correction of the FP multiple summation by storing and accomulating the cutoff digits in an intermediate number. Kahan method is described in the Appendix.

Tricks to get rid of the cancellation errors.

- rewrite to *avoid* cancellation:
  - **The quadratic root formula**: suppose $b > 0$ and $|4ac| \ll b^2$, then it is much better to rewrite

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-2c}{b + \sqrt{b^2 - 4ac}}.$$

  - **The iterated square-root**: we want to iterate $x \mapsto \sqrt{x}$. Doing it naïvely for positive initial x we would get 1 in a few steps, in default rounding mode (cf. Exercise 5). How to do it better? Write $x = y + 1$, then $y \mapsto \sqrt{y+1} - 1$. Now use the preceding trick to write $\sqrt{y+1} - 1 = y/(\sqrt{y+1} + 1)$.
- rewrite to *compensate* cancellations: consider the example of $\ln(x + 1)/x$. When $x < \epsilon/2$ we have $1 \oplus x = 1$ and, therefore, $LN(x \oplus 1) \oslash x = 0$ instead of the correct 1! The idea here is to notice that since $\ln(x + 1)/x \to 1$ as $x \to 0$, we make only a small error if we replace x by a little bit different value. Now comes the trick: replace x by $\bar{x} = (x \oplus 1) \ominus 1$ (compiler, please don't optimize this!). Then $LN(\bar{x} \oplus 1) \oslash \bar{x}$ is a good formula, unless $\bar{x} = 0$, in which case we take simply 1 as the result. Notice that since $\bar{x} \oplus 1 = x \oplus 1$, we can more simply write $LN(x \oplus 1) \oslash \bar{x}$. Interestingly, at the face value this formula looks like round-off/round-off, but actually it is very accurate!

Exercise 5 (Limit of iterated square-root) *We iterate in binary floating-point arithmetic the recursion formula $x_{n+1} = SQRT(x_n)$. We suppose that $SQRT$ is exactly rounded and that x_0 is a positive normal number. Does the sequence (x_n) have a limit as $n \to \infty$ and if yes what is it ? Give the answer for all four IEEE 754 rounding modes.*

Exercise 6 (Accuracy of iterated rewritten square-root) *Suppose you do the computations of the rewritten square-root iteration formula in double precision, starting from some value $x > 1$, and you do 100 iterations. How accurate is the answer ? What would you do if you had to start from any normal machine representable value $0 < x < 1$?*

Round-off errors are amplified by singularities in the function to be calculated:

- spurious singularities: appears only in the specific formula (in some sub-expression) used to evaluate the function, although the function itself has no singulaities → rewrite the formula to eliminate problem; not always obvious to see that there is a problem.
- real singularity: $\ln x$, $1/x$, arcsin.

Even small errors can be fatal if:

- some application may rely on monotonicity of a function: uncontrolled round-off errors can break this monotonicity slightly;
- When evaluating $f(g(x))$, it may happen that $g(x)$ falls ever so slightly outside of the domain of f: eg. take $x > 0$, then

  $$\sqrt{(((x \oplus 1) \ominus 1) \ominus (x \oslash 2))}$$

  will give a NaN when $x < \epsilon/2$, although the underlying mathematical formula would give a perfectly real result.

Exercise 7 (Round-off destroys monotonicity) *Come up with a simple example of round-off destroying monotonicity.*

How many significant digits does a calculation need?

Let us examine cases showing that, contrary to intuition, you may need to do calculations with much more significant digits than that implied by the input data (and maybe even then you can fail):

Take as our first example the function $f(x) := (\tan(\sin x) - \sin(\tan x))/x^7$ (note the Taylor development of f around $x = 0$, easily obtained by MAPLE for example: $f(x) = 1/30 + (29/756)x^2 + O(x^4)$, and then $f'(x) = (29/378)x + O(x^3)$). If its argument $x = 0.0200000$ is accurate to 6 significant decimals, how accurately does it determine $f(x)$ and how much precision must arithmetic carry to obtain that accuracy from the given expression? This x determines $f(x) = 0.0333486813\,(\simeq 1/30)$ to about 9 sig. dec. ($f'(0.02) \simeq 29/378 \times 0.02 \simeq 1.5 \cdot 10^{-3}$) but at least 21 must be carried to get that 9 (since $\tan(\sin 0.02) \simeq \sin(\tan 0.02) \simeq 0.02$ and $0.02^7 = 1.28 \cdot 10^{-12}$, so we want 9 significant decimals in the difference of two numbers that differ by about 1 part in 10^{12}, and we conclude $9 + 12 = 21$; MAPLE permitting, I have done the calculations and effectively we get the correct answer at 21 digits, but not with fewer digits).

Ever increasing precision usually works, but it can be slow. And it is certainly not a universal cure. For example, for real variables x and z define three continuous real functions E, Q and H thus: $E(z) :=$ if $z = 0$ then 1 else $(\exp(z) - 1)/z$; $Q(x) := |x - \sqrt{x^2 + 1}| - 1/(x + \sqrt{x^2 + 1})$; $H(x) := E(Q(x)^2)$. Then letting $x = 15.0, 16.0, 17.0, \ldots, 9999.0$ in turn compute $H(x)$ in floating-point arithmetic rounded to the same precision in all expressions. No matter how high the precision, the computation almost always delivers the same wrong $H(x) = 0$. Try it! In perfect arithmetic $Q(x) = 0$ instead of roundoff, so the correct answer is $H(x) = 1$.

Exercise 8 (Infinite precision is not enough) *$Q(x)$ is the difference of two terms which are algebraically equal, but computer arithmetic is not exact algebra. Take x to be an integer in the range $15.0, \ldots, 1000.0$. Estimate the absolute error of $x \pm \sqrt{x^2 + 1}$. Which is the bigger relative error ? Estimate now the absolute errors of each of the two terms of which Q is the difference. Which of the two has typically bigger absolute error ? Now give an estimate for Q. Why is then $H(Q(x)^2) = 0$?*

Extra reading: Base 10

Why use base 10, as proposed by IEEE 754r? Because financial computations for example cannot be correctly done in binary, since

- they involve 0.1, 0.01, etc., numbers which don't have a finite representation in binary since 5 (a factor of 10) and 2 are coprimes. If we tried to compute in binary floating-point, we would have round-off errors in the conversion deciaml to binary and backwards. This can compromise the results.

- the laws require round-off in decimal (eg. fixed conversion procedure between Euros and national currency)

In fact, almost 99% of financial software uses decimal floating-point. However without machine support, this is much (100 to 1000 times) slower than binary.

[4] Do applications actually use decimal data?

Yes. Data collected for a survey[5,6] of commercial databases analyzed the column datatypes of databases owned by 51 major organizations. These databases covered a wide range of applications, including Airline systems, Banking, Financial Analysis, Insurance, Inventory control, Management reporting, Marketing services, Order entry, Order processing, Pharmaceutical applications, and Retail sales.

In all, 1,091,916 columns were analysed. Of these columns, 41.8% contained identifiably numeric data (53.7% contained 'char' data, with an average length of 8.58 characters, some of which will have contained numeric data in character form).

Of the numeric columns, the breakdown by datatype was:

Type	Columns	percent
Decimal	251038	55.0
SmallInt	120464	26.4
Integer	78842	17.3
Float	6180	1.4

These figures indicate that almost all (98.6%) of the numbers in commercial databases have a decimal or integer representation, and the majority are decimal (scaled by a power of ten). The integer types are often held as decimal numbers, and in this case almost all numeric data are decimal.

Appendix: Kahan summation algorithm (from Wiki)

In numerical analysis, the Kahan summation algorithm minimizes the error when adding a sequence of finite precision floating point numbers.

It is of basic use, therefore, on computers. It is also called compensated summation. As the name suggests, this algorithm is attributed to William Kahan.

[4] from http://www2.hursley.ibm.com/decimal/decifaq1.html

[5] "A Study of DataBase 2 Customer Queries", Annie Tsang and Manfred Olschanowsky, IBM Technical Report TR 03.413, 25pp, IBM Santa Teresa Laboratory, San Jose, CA, April 1991.

[6] see also: page 11 of "How Customers use DB2 for z/OS", June 2004, Chris Crone, Manfred Olschanowsky, Akira Shibamiya; available from ftp://ftp.software.ibm.com/software/db2storedprocedure/db2zos390/techdocs/catsurvey_ext.pdf

In pseudocode, the algorithm is:

```
function kahanSum(input, n)
    var sum = input[1]
    var c = 0.0          //A running compensation for lost low-order bits.
    for i = 2 to n
    y = input[i] - c      // So far, so good: c is zero.
    t = sum + y           // Sum is big, y small, so low-order digits of y are lost.
    c = (t - sum) - y     //(t - sum) recovers the high-order part of y;
                          // subtracting y gives the the low part of y
    sum = t               // Algebriacally, c should always be zero.
    next i                // Next time around, the lost
    return sum            // low part will be added to y in a fresh attempt.
```

An example in six-digit floating-point decimal arithmetic. Suppose Sum has attained the value 100000 and the next value of input(i) is 3.14159 (a six-digit floating point number) and c has the value zero.

$y = 3.14159 - 0$	
$t = 100000 + 3.14159$	
$= 100003$	Many digits have been lost!
$c = (100003 - 100000) - 3.14159$	This must be evaluated as written!
$= 3.00000 - 3.14159$	The assimilated part of y recovered, vs. the original full y.
$= -0.141590$	The trailing zero because this is six-digit arithmetic.
$sum = 100003$	Thus, few digits from input(i) met those of the sum.

The sum is so large that only the high-order digits of the input numbers are being accumulated. But on the next step, suppose input(i) has the value 2.71828, and this time, c is not zero...

$y = 2.71828 - -0.141590$	The shortfall from the previous stage has another chance.
$= 2.85987$	It is of a size similar to y: most digits meet.
$t = 100003 + 2.85987$	But few meet the digits of sum.
$= 100006$	Rounding is good, but even with truncation,
$c = (100006 - 100003) - 2.85987$	This extracts whatever went in.
$= 3.00000 - 2.85987$	In this case, too much.
$= .140130$	But no matter, the excess will be subtracted off next time.
$sum = 100006$	

So the summation is performed with two accumulators: sum holds the sum, and c accumulates the parts not assimilated into sum, to nudge the low-order part of sum the next time around. Thus the summation proceeds with "guard digits" in c which is better than not having any but is not as good as performing the calculations with double the precision of the input. However, if input is already in double precision, few systems supply quadruple precision, and if they did, what if input were quadruple precision...

Another approach is to perform the summation on differences from a working mean (in the hope that the value of sum never becomes much larger than individual differences), except that the values might be quite different from the working mean and thus suffer significant truncation. Alternatively, sort the values and pair positive and negative values so that the accumulated sum remains as close to zero as possible, at great cost in computational effort.

[edit] Caution! Beware Optimising Compilers!

The code produced by the compiler must follow the steps exactly as written! Alas, writers of compilers are often casual in their understanding of floating-point arithmetic and when considering optimisations are seduced by the purity of the mathematical analysis of real numbers when instead it is floating-point arithmetic that is being dealt with. Applied to this code

$$t := sum + y;$$

$$c := (t - sum) - y;$$

it will swiftly be deduced that

$$c = 0$$

which is constant and need not be computed within the loop; further, since it is initialised to zero, the statement

$$y := input[i] - c;$$

can be contracted so that the loop becomes

$$y := input[i];$$

$$t := sum + y;$$

$$sum := t;$$

and further that these variables are just waystations, so

$$sum := sum + input[i];$$

It might be that some particular optimising compiler would carry its analyses so far as to deduce that a summation of input is intended, and then generate

code employing maximum precision and its own version of this algorithm, but far more likely is that it will do something that will result in code that wrecks the workings of the algorithm you have coded.

The algorithm's execution can also be affected by non-mathematical optimisations. For instance, it is quite common for the floating-point computations to be carried out in machine registers that have a precision higher than that of the variables held in main storage, as in the IBM-PC and clones where the registers hold 80-bit floating-point numbers while in main storage they might be held only as 32-bit, or 64-bit as well as 80-bit. The sequence

$$y := input[i] - c;$$

$$t := sum + y;$$

$$c := (t - sum) - y;$$

might be compiled without any of the unwanted mathematical transformations, but, notice that after the code for the first statement is executed, the register holding the result that was stored in y still has (or could still have: the registers might be organised as a stack with overflow to memory) that result and as the next statement refers to y, perhaps the code for it could be arranged so that the value of y need not be fetched from memory; similarly for t in the third statement. If the stored values are held to the same precision as the registers, then all will be well: the registers and main storage are thus equivalent except for the speed of access. But if not, the working of the algorithm will again be ruined. Optimisation options helpful for some parts of the programme will not necessarily be good for all parts of a programme.

It might be that some optimising compiler will recognise that variables y and t are waystations that need not be saved into main memory (especially if their usage is only within the loop), and it may further be that the computer has enough floating-point registers available that the compiler can generate code that employs only full-precision registers to hold sum and the waystations (saving the summation into sum at the end of the loop), but input may be a complex entity such as a computation itself needing registers for its evaluation, too many registers, or worse still, be a function invocation with an arbitrary requirement. Thus it is unlikely that the optimum outcome of a full-precision summation via registers employing this algorithm will be attained without the most careful attention to detail, even inspection of the machine code produced.