Machine-independent Algorithms for floor(x) and ceil(x)

W. Kahan
E. E. & C. S. Dept.
Univ. of Calif.
Berkeley

floor(x) := the largest integer no larger than x ;
ceil(x) := −floor(−x) , for all real x .

Can these functions be difficult to compute? Apparently they are difficult enough to program that one major player in the computing world charges a stiff fee for use of the company's programs. On a machine that does not conform to IEEE standard 754 or 854 for floating-point arithmetic, or on a machine that does conform but whose compiler doesn't, computing these functions can be an interesting challenge. The challenge must be met without using INTEGER arithmetic because the computer may well lack an INTEGER data-type as wide as its widest REAL (floating-point) type. And an assembly-language program is no good because it cannot be moved to any other computer, with the rest of the software in which it is embedded, by mere recompilation. The challenge must be met with a program written in a higher-level compiled language which, like C , lacks these intrinsic functions.

The trouble with the IEEE standards 754 and 854 is that they require capabilities that may well be provided by hardware and yet be inaccessible from a higher-level language for lack of standard names for those capabilities. Here is an algorithm to compute floor(x) and ceil(x) quickly on a standard-conforming machine; see whether you can program it in your favorite language:

> Save the rounding-direction mode;
> Set that mode to Round to +∞ for ceil,
> Round to −∞ for floor ;
> Round (Convert) x to an integer value;
> Restore the former rounding-direction mode.

If we must compute floor and ceil using only the rudimentary rational operations and comparisons available in all higher-level languages, and do so in a way that recompiles and runs correctly on *all* commercially significant computers, this simple problem grows into a monster. We have to exploit properties common to all floating-point arithmetics, regardless of how they are rounded; such properties are not obvious. Here are the ones we need:

The REAL Constant Λ .

All sufficiently large floating-point numbers are integers. (In fact, all sufficiently large floating-point numbers are *even* integers; taking this to the limit suggests that ∞ is an even integer too, or nearly enough so for government work.) Therefore each computer has its constant Λ = 1000...000, the smallest REAL number such that every REAL $x \ge \Lambda$ must be an integer too. Λ varies from machine to machine, but it can be computed in a way to be discussed later.

Λ has several exploitable properties. First, the consecutive integers $\Lambda, \Lambda+1, \Lambda+2, \ldots, 2\Lambda$ constitute *all* the REAL numbers between Λ and 2Λ inclusive, whereas $\Lambda - 1/2$ is one of the REAL numbers lying between Λ and $\Lambda-1$. Any attempt to compute a non-integer real value ξ between Λ and 2Λ must encounter at least one rounding error; if the value ξ is rounded just once to x, as might occur when ξ is the result of a single *add* operation, then we can expect to find $|x - \xi| < 1$ on all commercially significant machines. (This error bound might not be valid for other operations like *subtract* or *multiply* or *divide* on certain machines, for instance on CRAYs; fortunately, we rely only on the accuracy of *add* .) Certain floating-point operations always execute *exactly* on all commercially significant machines. If x is REAL and lies in the interval $\Lambda \le x \le 2\Lambda$ then $x - \Lambda$ will be exact, and if $1 \le x < 2\Lambda$ then $x - 1$ must be exact, and so must $x + 1$ if x has an integer value. *Comparison* ($x<y$, $x=y$ and $x>y$) and *Negation* ($-x$) are assumed exact too despite that compilers on CDC Cybers have been known to violate the first assumption; such a violation seems more like a bug than a feature to be encouraged.

Computing floor and ceil .

Here is an algorithm to compute REAL floor(x) and ceil(x) for any REAL x ; it uses one REAL scratch variable y .

```
    If  x < 0  then  return  floor(x) := -ceil(-x)   and
                             ceil(x)  := -floor(-x) .
    If  x ≥ Λ  then return  floor(x) := ceil(x) := x .
               ...  Now  0 ≤ x < Λ .
    y := (Λ + x) - Λ ;  ... an integer,  and  |x - y| < 1 .
    If  x = y  then  return  floor(x) := ceil(x) := x
       else if  x < y  then return  floor(x) := y-1  and  ... ***
                                    ceil(x)  := y        ... ***
             else  return  floor(x) := y  and  ceil(x) := y+1 .
    End. ...  The two statements marked  ***  are not necessary on
         ...  CRAYs nor IBM 370s  because their adds are chopped.
```

This algorithm can be thwarted if the scratch variable y resides in a register carrying more precision than REAL variables like x for which Λ was determined, so make sure that x, y and Λ are declared to have the widest REAL type supported by the hardware. If the compiler pays no attention to parentheses, separate the statement " $y := (\Lambda + x) - \Lambda$ " into two statements.

What should be done on a CDC Cyber if *Comparison* is suspect? The following suggestions are offered not to legitimize defective compilers but to permit programmers generally to get on with life: A few changes suffice. Change " $x \ge \Lambda$ " to " $x-\Lambda/2 \ge \Lambda/2$ " , " $x = y$ " to " $x - 0.5 = y - 0.5$ " , and insert a statement

```
  "  If  0 < x  and  x-0.5 < 0.5  then return  floor(x) := 0  and
                                               ceil(x)  := 1 . "
```

and a comment " ... Now $x = 0$ or $1 \le x \le \Lambda$." in place of of the comment " ... Now $0 \le x \le \Lambda$." These changes do no harm to other computers except for a loss of speed and perspicuity. If the Cyber 1xx's compiler emits chopped FX instead of pseudo-rounded RX floating-point operations, then the two statements marked *** can be omitted.

What is Λ ?

The value of Λ should be determined once for each compiler on each machine, rather than every time floor or ceil is invoked. A table of values for various machines' floating-point hardwares is supplied below. However, a program cannot be expected to read that table; if the program is to be completely portable at the cost solely of recompilation, without the need for knowledgeable intervention to supply a plethora of installation-time parameters, then the program must somehow compute Λ once and save it for subsequent reuse. In fact, such a computation may be the only way to defend against mistaken values of Λ supplied either by faulty Decimal-to-Binary conversion programs, or by people who claim to be knowledgeable but aren't knowledgeable enough. " A little knowledge is a dangerous thing; ... ."

Two ways to compute Λ are presented here so that they may be compared for consistency; discrepancies call urgently for human intervention. For instance, computers have been built whose every REAL number is represented by its sign and the logarithm of its magnitude; since at most five consecutive integers can be represented exactly as REALs on such a machine, the operations floor and ceil become dubious. Other computer arithmetics have been proposed (but not yet built into any North American machine as far as I know) that divide each REAL word in memory into two variable-width fields for exponent and significant digits; these require that Λ be chosen in a way that takes account of internal registers used by the compiler but inaccessible to the programmer. Both of these unusual arithmetics will generate discrepant results from the two programs below. Were Λ determined just once, as in the program MACHAR provided by W. J. Cody and W. Waite in their *Software Manual for the Elementary Functions* (Prentice-Hall, 1980), no warning could emerge.

TABLE OF VALUES OF Λ FOR A FEW MACHINES

Machine	Format	Λ
IBM 370	REAL*4	16^{5} = 1048576.
	REAL*8	16^{13} = 4503599627370496.
	REAL*16	16^{27} = 324518553658426726783156020576256
DEC VAX	REAL*4 (F)	2^{23} = 8388608.
	REAL*8 (G)	2^{52} = 4503599627370496.
	REAL*8 (D)	2^{55} = 36028797018963968.
	REAL*16 (H)	2^{112} = 5192296858534827628530496329220096
CDC Cyber	REAL 60 bit	2^{47} = 140737488355328.
CRAY	REAL 64 bit	2^{47} = 140737488355328.
IEEE 754	SINGLE	2^{23} = 8388608.
	DOUBLE	2^{52} = 4503599627370496.
	EXTENDED 80 bit	2^{63} = 1152921504606846976.

Among the machines that have these three formats are those that use the Motorola 68881, e.g. the SUN III and Apple Macintosh, or the Intel 8087/80287/80387, e. g. IBM's PC, XT, AT but not RT, or the AT&T WE32106. The HP Spectrum series EXTENDED format has the same Λ as the DEC VAX H format. Floating-point chips made by National, AMD, TI, WEITEK and BIT support at most the SINGLE and DOUBLE formats in, e.g., IBM's RT-PC.

One way to compute Λ = 1000...000 is to compute the arithmetic's radix B = 10 first; this means *two* on binary machines, *eight* on octal, *ten* on decimal and *sixteen* on hexadecimal machines. Then $\Lambda = B^{P-1}$ where P is the number of significant B-digits carried. The algorithm offered here is derived from one of Mike Malcolm's (Comm. ACM v. 15, 1972) but modified in a way that has worked, in the author's PARANOIA program, on a wide range of machines except perhaps only the Cyber 2xx series (with 64-bit words) and its ETA cousins with certain compilers.

```
One := REAL(1) ;  Two := One + One ;  Zero := One - One ;
Mone := -One ;
If  ( One=Zero or One*One+Mone≠Zero or One-Two≠Mone ) then
                         print "Now who's paranoid?"  and  Quit.
w := One ;
Do {  w := w + w ;  u := | ((w+One) - w) - One |
   } until  u + Mone ≥ Zero ;
...  Now  w = 2^k  is just big enough that  |((w+1)-w)-1| ≥1 .
u := One ;
Do {  B := (w + u) - w ;  u := u + u
   } until  B > Zero ;  ...  Now  B  is the  Radix.
If  B < Two  then print "A logarithmic machine!"  and Quit.
w := One ;
Do {  Λ := w ;  w := B*w ;  u := (w + One) - w
   } until  u ≠ One ;  ...  Now  Λ  is known.
```

The second way to compute Λ , and to corroborate the first, is also drawn from the author's PARANOIA program described in *BYTE* 10 #2 (Feb. 1985, pp. 223-235) by R. Karpinski. The idea is to find out fast how 1.0 differs from the next larger REAL number; that difference should be 1/Λ unless the widths of the fields of a floating-point number vary with its magnitude.

```
Four := Two + Two ;  Three := Two + One ;  HexD := Four*Four ;
v := Four/Three - One ;         ...  v is very near 1/3 .
w := | ((v+v) - One) + v | ;    ...  w = 3*|error in 4/3| .
If  w = Zero  then
       print "Ternary arithmetic?  Not in the USA !"  and Quit.
Do {  e := w ;  w := ((HexD*w*w + w/Two) + One) - One
   } until  ( w ≥ e  or  w = Zero ) ;  ...  Now  e = 1/Λ .
If  Λ*e ≠ One  then print "Λ  may be wrong!"  and  Stop.
```

Both algorithms above can be ruined by compilers that disregard parentheses; for such compilers, break statements in such a way as will force the desired order of evaluation. Both algorithms are designed to determine Λ correctly even if intermediate expressions are evaluated in registers with more precision than REAL variables have in memory, but then only if parentheses are honored by the compiler.

Epilogue

The problem of computing floor and ceil in a completely portable way without reliance upon someone else's proprietary software nor upon manually inserted constants nor upon unreliable compilers nor upon idiosyncratic hardware is not a problem invented just for the classroom. The problem was presented to the author by a colleague (Prof. John Ousterhout) in all seriousness. But it is still an unreasonable problem; applications programmers should not have to solve it over and over again. We ought to be able to depend upon a library of mathematical functions supplied with each machine by its maker and used consistently by all compilers of all languages for that machine. The *SANE* Standard Apple Numerical Environment described in *Standard Apple Numeric Environment for All Macintosh and Apple II Computers* (Addison-Wesley, 1986, with a new edition to appear immenently) is a good example of what we all need. The DEC VAX VMS Fortran library would be another good example were it freely available to users of UNIX on VAXes too. Such a library would supply computer users with a rich collection of mathematical functions that would, ideally, be accessible in all languages and available on all computers, though the precise values of those functions might have to vary a little from machine to machine even if all their arithmetics conformed to a standard like IEEE 754 . For a readable description of that standard see "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic" by W. J. Cody *et al*. in the IEEE magazine *MICRO* for Aug. 1984, pp. 86-100. An earlier paper by the author and J. T. Coonen, "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments" in *THE RELATIONSHIP BETWEEN NUMERICAL COMPUTATION AND PROGRAMMING LANGUAGES* edited by J. K. Reid (North-Holland, 1982), advocated a computing environment throughout which a universal library of mathematical functions could more easily be disseminated despite persistent variance in the semantics of computer arithmetic.

To reach the desired state of affairs we need a standard for the names and specifications for the functions in that library. Silly naming inconsistencies among languages will have to persist just for the sake of compatibility with prior practice; an example is BASIC's use of SQR for what everyone else calls SQRT ($\sqrt{(x)}$) while Pascal uses SQR for x^2 , the inverse of SQRT. Such a standard should not be left to language enthusiasts alone because they will give too much weight to implementation problems that they are ill equipped to handle, too little weight to the needs of applications programmers. For similar reasons, most machine manufacturers are not eligible. Producers and users of portable numerical software must preponderate.

Now, who shall bell the cat?

Acknowledgments
Much of the author's work along these lines has been supported by a long-running grant from the U. S. Office of Naval Research.