A Comment on the Eispack Machine Epsilon Routine

Tim Hopkins and John Slater
Computing Laboratory
University of Kent
Canterbury, CT2 7NF
Kent, UK.

March 26, 2001

Abstract

We analyze the algorithm used to generate the value for the machine epsilon in the Eispack suite of routines and show that it can fail on a binary floating-point system. The comments in the code describing the conditions under which this method will work are not restrictive enough and we provide a replacement set of assumptions. We conclude by suggesting how the algorithm may be modified to overcome most of the shortcomings.

1 Introduction

A number of algorithms designed to compute the machine epsilon (the smallest representable number, ϵ, such that $1 + \epsilon > 1$ using the machine's floating-point arithmetic) have appeared over the years see, for example, [**?**], [**?**] and [**?**]. Perhaps the simplest forms part of the Eispack suite of eigensystem routines ([**?**] and [**?**]). The Fortran code implementing this algorithm is given in figure **??**. Comments which appeared in the original routine claim that the code will function properly on all systems satisfying the two following assumptions

1. the base used in representing the floating-point numbers is not a power of three,
2. the quantity a in statement 10 is represented to the accuracy used in the floating-point variables that are stored in memory.

Under these assumptions it is asserted that the following statements should be true

1. a is not exactly equal to four-thirds,
2. b has a zero for its last digit,
3. c is not exactly equal to one,
4. eps measures the separation of 1.0 from the next larger floating point number.

```
      double precision function epslon (x)
      double precision x
c
c     estimate unit roundoff in quantities of size x.
c
      double precision a,b,c,eps
c
c     this version dated 4/6/83.
c
      a = 4.0d0/3.0d0
   10 b = a - 1.0d0
      c = b + b + b
      eps = dabs(c-1.0d0)
      if (eps .eq. 0.0d0) go to 10
      epslon = eps*dabs(x)
      return
      end
```

Figure 1: Fortran code implementing the Eispack machine epsilon algorithm

2 An Example

It is instructive to look in some detail at how this algorithm works. Assume

1. x is unity,
2. the base of the arithmetic is 2,
3. the floating-point mantissa is normalized, (i.e., it is of the form $1.f$),
4. The mantissa is represented by an odd number of bits; this may include a hidden bit.

Thus f has an even number of bits and

$$\begin{aligned} a = 1.f &= 1.01010\cdots0101 \times 2^0 \\ b = a - 1.0 &= 0.01010\cdots0101 \times 2^0 \\ &= 1.01010\cdots0100 \times 2^{-2} \end{aligned}$$

The subtraction of 1.0 preserves the repeated pattern of bits in the mantissa with the exception of the last bit which is now zero due to the left shift. Whence

$$\begin{aligned} b + b &= 10.10\cdots1000 \times 2^{-2} \\ &= 1.010\cdots0100 \times 2^{-1} \end{aligned}$$

and adding a further b gives

$$b + b + b = 1.1111\cdots110 \times 2^{-1}$$

Subtracting the result from 1.0 we obtain $0.000\cdots001 \times 2^0$ which is the machine epsilon as required.

Note that the algorithm works correctly no matter whether the floating-point addition rounds or truncates.

3 A Failure

Assume now that the mantissa stores an even number of bits (i.e., f has an odd number of bits) and that division truncates, a and b thus have the form

$$\begin{aligned} a = 1.f &= 1.01010\cdots 1010 \times 2^0 \\ b = a - 1.0 &= 0.01010\cdots 1010 \times 2^0 \\ &= 1.01010\cdots 1000 \times 2^{-2} \end{aligned}$$

whence

$$b + b + b = 1.1111\cdots 1100 \times 2^{-1}$$

and subtracting the result from 1.0 we obtain

$$eps = 0.000\cdots 010 \times 2^0$$

which is a factor of 2 too large. Once again the algorithm will deliver this result whether the subsequent arithmetic rounds or truncates. Note that, if $a = 1.f$ is correctly rounded to $1.01010\cdots 1011 \times 2^0$, the correct value of ϵ is obtained.

4 The General Case

The above case is not the only one that causes failure. To analyse the algorithm further we first need to obtain the different forms that the decimal expansion of $1/3$ may take for a general base, β. Defining $r = \lfloor \beta/3 \rfloor$ and $s = 2r + 1$ there are three cases

1. $\beta \bmod 3 = 1$: $\quad (1/3)_\beta = .\dot{r}$,
2. $\beta \bmod 3 = 2$: $\quad (1/3)_\beta = .\dot{r}\dot{s}$,
3. $\beta \bmod 3 = 0$: $\quad (1/3)_\beta = .r$

For case **??** we see that assumption 1 is not strong enough to ensure that a is not exactly equal to four-thirds provided the mantissa is of length at least 2. Running the routine *epslon* with such a base arithmetic would result in an infinite loop!

Consider the case $\beta \bmod 3 = 2$ and assume that division truncates the result, then, depending on whether an odd or even number of significant digits are stored, b may take the form

$$r.srs\cdots rs0 \times \beta^{-1} \quad \text{or} \quad r.srs\cdots sr0 \times \beta^{-1}$$

Whence, noting that $\beta = 3r + 2$ and $2\beta = 3s + 1$, $b + b$ equals

$$s.rsr\cdots sr0 \times \beta^{-1} \quad \text{or} \quad s.rsr\cdots r(s-1)0 \times \beta^{-1}$$

and, defining $\gamma = \beta - 1$, $b + b + b$ is given by

$$\gamma.\gamma\gamma\cdots\gamma\gamma 0 \times \beta^{-1} \quad \text{or} \quad \gamma.\gamma\gamma\cdots\gamma(\gamma-1)0 \times \beta^{-1}$$

This leads to the correct value for ϵ if an odd number of significant digits are stored and a value of $2 \times \epsilon$ if an even number of digits are stored.

If the result of a division is correctly rounded to an even number of digits then b is of the form

$$r.srs\cdots s(r+1)0 \times \beta^{-1}$$

whence $b+b+b = 1.00\cdots 01 \times \beta^0$ and the correct value for ϵ is obtained.

The correct value for ϵ is also obtained when $\beta \bmod 3 = 1$.

Without some prior knowledge of β it is not possible to guarantee that the routine will work in the other cases. However, we note that when β is odd the expansion of $(1/2)_\beta$ is of the form $0.\dot{t}$ where $t = \lfloor \beta/2 \rfloor$. Substituting the lines

```
      a = 3.0d0/2.0d0
   10 b = 1 - 1.0d0
      c = b + b
```

for

```
      a = 4.0d0/3.0d0
   10 b = 1 - 1.0d0
      c = b + b + b
```

will then generate the correct value of ϵ. For the case $\beta \bmod 3 = 2$, β even and $\beta \neq 2$ we may obtain an expansion of the form $0.\dot{p}$ by using $(\beta/(\beta-1))_\beta$ although this does not help with the search for a general implementation.

5 Conclusion

For the code given in **??** the assumptions given in §**??** need modifying to

1. the base used in the representing the floating-point numbers is not a multiple of three,
2. the result of a floating-point division is correctly rounded or, if it is truncated, the mantissa stores an odd number of digits.

The second assumption may be taken into account if the algorithm is changed to take account of the possibility that the returned value may be double the required result. This would mean replacing the line

```
      if (eps .eq. 0.0d0) go to 10
```

by

```
      if (eps .eq. 0.0d0) then
print *, "Failure: arithmetic base may be a multiple of 3"
stop
      else
c eps may be out by a factor of two
eps = eps/2.0d0
if (1.0d0 + eps .eq. 1.0d0) then
  eps = 2.0d0 * eps
endif
      endif
```

The above code would need to be modified to prevent clever compilers from keeping variables in extended precision registers or finding ways of subverting the test `(1.0d0 + eps .eq. 1.0d0)`.

The best way of providing a portable means of obtaining the machine epsilon would seem to be that provided by Fortran 90; the intrinsic function EPSILON returns the required result!