Math 4610 Lecture Notes

Machine Epsilon Code [*]

Joe Koebbe

August 23, 2019

[*]These notes are part of an Open Resource Educational project sponsored by Utah State University

Math 4610: Development of a Test for Machine Precision.

To determine machine precision for your computer we need a strategy that will display accuracy. We know that the number of digits kept in a machine precision number is finite. In addition, we know that the numbers are stored in a binary number system. As scientists, engineers, and computer scientists the results that we want to see are decimal (base 10) numbers. Our goal should be to determine the resolution of our numbers in base 10. One way to proceed would be to try to compare a sequence of numbers converging to zero until the machine numbers are internally no different than zero. The problem with this strategy is that we will need to compute the difference between a term in the sequence and the number zero. That is,

```
    error = number - zero
```

The trouble with this approach is that in the difference of the two numbers that are close to zero may cancel and in floating point arithmetic, there is no guarantee as to what values will be returned due to the rounding algorithm used by any computer. That is, if digits that are beyond the finite representation of the computer numbers are included in the computation, there is no guarantee the values returned. This is the old computer programming adage of garbage in equals garbage out.
A more stable approach is to compare a sequence of numbers converging to one meaning consider

```
     error = 1 - ( 1 + epslion )
```

where epsilon is represents a decreasing sequence of values. Note also that the number one is represented exactly in binary arithmetic. Notice that there are parentheses around the last two terms. This will force a code to evaluate the term in parentheses first before computing the difference.
So, let's write pseudo-code that will compute this difference as a parameter, ϵ, is made closer and closer to zero.

```
     // comment: initialize the constant one and the small value

     one = 1.0;
     eps = 1.0;

     // comment: loop over the computing the difference between 1 and 1 plus a
     //          bit

     for i=1,1000
       diff = one - ( one + eps );
       eps = 0.5 * eps;
     end
```

This will work, but there is no way to tell when the difference is zero. So, we need to add some way to identify the output.
So, let's add a line to print out the values we are interested in.

```
     // comment: initialize the constant one and the small value

     one = 1.0;
     eps = 1.0;

     // comment: loop over the computing the difference between 1 and 1 plus a
```

```
//          bit

for i=1,1000
  diff = one - ( one + eps );
  print diff, eps;
  eps = 0.5 * eps;
end
```

The last little issue is the number of times the algorithm will execute the loop is a bit high.
So, a last version of the algorithm might look like

```
// comment: initialize the constant one and the small value

one = 1.0;
eps = 1.0;

// comment: loop over the computing the difference between 1 and 1 plus a
//          bit

for i=1,1000
  diff = one - ( one + eps );
  print diff, eps;
  if(diff == 0.0) return;
  eps = 0.5 * eps;
end
```

The next thing we will need to do is translate this algorithm into code. This can be translated into C, C++, Fortran, Java, Python or any other language.
For the work at this point, the coding language used in the following example is fortran.

```
 subroutine smaceps(seps, ipow)
c
c set up storage for the algorithm
c --------------------------------
c
      real seps, one, appone
c
c initialize variables to compute the machine value near 1.0
c ----------------------------------------------------------
c
      one = 1.0
      seps = 1.0
      appone = one + seps
c
c loop, dividing by 2 each time to determine when the difference between
c one and the approximation is zero in single precision
c -----------------------------------------------------
c
      ipow = 0
      do 1 i=1,1000
         ipow = ipow + 1
c
c update the perturbation and compute the approximation to one
```

```
c ------------------------------------------------------------
c
        seps = seps / 2
        appone = one + seps
c
c do the comparison and if small enough, break out of the loop and return
c control to the calling code
c ---------------------------
c
        if(abs(appone-one) .eq. 0.0) return
c
    1 continue
c
c if the code gets to this point, there is a bit of trouble
c ---------------------------------------------------------
c
      print *,"The loop limit has been exceeded"
c
c done
c ----
c
return
end
```

In the Fortran code, there is a lot of documentation of the steps in the code. To compile the code in linux command terminal, the following will work if the compiler is installed.

```
% gfortran -o smaceps smaceps.f
```

The output from this compilation command is an object file named

```
maceps.o
```