Solutions for Homework 5 Foundations of Computational Math 1 Fall 2011

Problem 5.1

Consider the following numbers:

- 122.9572

- 457932

- 0.0014973

5.1.a. Express the numbers as floating point numbers with $\beta = 10$ and $t = 4$ using rounding to even and using chopping.

5.1.b. Express the numbers as floating point numbers with in single precision IEEE format using rounding to even. It is strongly recommended that you implement a program to do this rather than computing the representation manually.

5.1.c. Calculate the relative error for each number and verify it satisfies the bounds implied by the floating point system used.

Solution: The conversion to the decimal floating point is straightforward and is easily done by inspection.

$$122.9572 = .1229572 \times 10^3$$
$$\approx .1229 \times 10^3 \text{ chopped}$$
$$\approx .1230 \times 10^3 \text{ rounded}$$
$$457932 = .457932 \times 10^6$$
$$\approx .4579 \times 10^6$$
$$0.0014973 = .14973 \times 10^{-2} \approx .1497 \times 10^{-2}$$

For the IEEE single precision, the number of bits and the possibility of a nonterminating fraction complicates trying to do this by hand. However, the procedure you use is easily coded on an IEEE machine using double precision arithmetic to compute the single preceision representation. We illustrate one point of view on 122.9572 in detail. Consider first the nonfractional portion 122. We start with the largest power of 2 smaller than or equal to 122

$$f = 122$$
$$f \geq 2^6 = 64 \rightarrow b_6 = 1 \text{ and } f \leftarrow f - 2^6 = 58$$
$$f \geq 2^5 = 32 \rightarrow b_5 = 1 \text{ and } f \leftarrow f - 2^5 = 26$$
$$f \geq 2^4 = 16 \rightarrow b_4 = 1 \text{ and } f \leftarrow f - 2^4 = 10$$
$$f \geq 2^3 = 8 \rightarrow b_3 = 1 \text{ and } f \leftarrow f - 2^3 = 2$$
$$f < 2^2 = 4 \rightarrow b_2 = 0 \text{ and } f \leftarrow f$$
$$f \geq 2^1 = 2 \rightarrow b_1 = 1 \text{ and } f \leftarrow f - 2^1 = 0$$
$$f < 2^0 = 1 \rightarrow b_0 = 0 \text{ and } f \leftarrow f - 2^1 = 0$$
$$122 = b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$
$$= 2^6 + 2^5 + 2^4 + 2^3 + b_1 2^1 = 64 + 32 + 16 + 8 + 2$$
$$122 = (1111010)_2$$

The fractional part can also be converted to an expansion in terms of 2^{-i} by repeated comparison and subtraction. A code does this easily. When computed in double precision, all of the relevant negative powers of 2 needed for a single precision representation can be computed with enough extra to apply chopping or a rounding of your choice. The code below is essentially matlab code

Generates n fractional binary bits into $b(1:n)$

```
f=double(0.9572);
s=double(1.0);
n=40;
b(1:n)=0;
for k=1:n
    s=(s)/double(2.0);
    if f ≥ s
            b(k) = 1;
            f = f - s;
    end
end
```

Using this code on the fractional part 0.9572 to produce many more bits (the vertical line indicates the last mantissa bit) than is needed for single precision yields:

$$f = 122.9572$$
$$= 1.11101011110101000010110|0001111001001111 \times 2^6$$
$$\begin{bmatrix} \sigma \mid \epsilon \mid \mu \end{bmatrix} = \begin{bmatrix} 0 \mid 10000101 \mid 11101011110101000010110 \end{bmatrix}$$

Repeating the exercise on the other two numbers yields:

$$f = 457932 = 1.10111111001100110000000 \times 2^{18}$$
$$\begin{bmatrix} \sigma \mid \epsilon \mid \mu \end{bmatrix} = \begin{bmatrix} 0 \mid 10010001 \mid 10111111001100110000000 \end{bmatrix}$$
$$f = 0.0014973 = 1.10001000100000100001101000|100001 \times 2^{-10}$$
$$\begin{bmatrix} \sigma \mid \epsilon \mid \mu \end{bmatrix} = \begin{bmatrix} 0 \mid 01110101 \mid 10001000100000100001101001 \end{bmatrix}$$

Problem 5.2

Consider the function
$$f(x) = \frac{1.01 + x}{1.01 - x}$$

5.2.a. Find the absolute condition number for $f(x)$.

5.2.b. Find the relative condition number for $f(x)$.

5.2.c. Evaluate the condition numbers around $x = 1$.

5.2.d. Check the predictions of the condition numbers by examining the relative error
and the absolute error

$$err_{rel} = \frac{|f(x_1) - f(x_0)|}{|f(x_0)|}$$
$$err_{abs} = |f(x_1) - f(x_0)|$$

with $x_0 = 1$, $x_1 = x_0(1 + \delta)$ and δ small.

Solution:
We have for a slightly more general problem

$$f(x) = \frac{\gamma + x}{\gamma - x}$$
$$f'(x) = \frac{2\gamma}{(\gamma - x)^2}$$
$$\kappa_{rel} = \frac{|x f'(x)|}{|f(x)|} = \frac{|2\gamma x|}{|(\gamma - x)(\gamma + x)|}$$
$$\kappa_{abs} = |f'(x)| = \frac{|2\gamma|}{|(\gamma - x)^2|}$$

So as $x \to \gamma$ the conditioning worsens.

Let $\gamma = 1.01$ and $x_0 = 1$ then

$$f(1) = 201.00$$
$$f'(1) \approx 2 \times 10^4 = \kappa_{abs}$$
$$\kappa_{rel} \approx 10^2$$
$$\delta = 10^{-5} \rightarrow f(1 + \delta) = 201.2$$
$$f(1 + \delta) - f(1) = 0.2 = 2 \times 10^4 \times \delta$$
$$\frac{f(1 + \delta) - f(1)}{f(1)} = \frac{0.2}{201.0} \approx 10^{-3} = \kappa_{rel}\delta$$

So the predictions are accurate.

Problem 5.3

Most recent machines use a binary base, $\beta = 2$, but the number of bits, t, may vary in a floating point system. The following algorithm is an attempt to determine t experimentally.

$x = 1.5$, $u = 1.0$, $t = 0$, $\alpha = 1.0$
while $x > \alpha$
    $u = u/2$
    $x = \alpha + u$
    $t = t + 1$
end

5.3.a. Prove that this algorithm finds t for a floating point system with $\beta = 2$ or explain why it does not.

5.3.b. Apply the algorithm to a machine that uses $\beta = 2$ in single precision to find t_s.

5.3.c. Apply the algorithm to a machine that uses $\beta = 2$ in double precision to find t_d.

5.3.d. Do the numbers t_s and t_d agree with the IEEE floating point standard for single and double precision floating point numbers?

Solution: This is a well known code that can be used as a simple example of an automatic method to determine the floating point characteristics of a machine. We assume we are operating on a machine with $\beta = 2$. Therefore, 1 and 2^{-k} are representable exactly with k in the range of the number of bits of the mantissa. Futhermore, we assume that a hidden bit normalization is used with weight 2^0. Since we are working with numbers around 1 we can ignore the exponent in the floating point representation. If the machine had $t = 4$ and used round-to-even based extra rounding bits kept during the addition (below assumes three such bits and they are displayed after the vertical line) we would have the following internal representations of x before and after rounding.

4

iteration	before rounding	after rounding	t printed
1	1.1000|000	1.1000	1
2	1.0100|000	1.0100	2
3	1.0010|000	1.0010	3
4	1.0001|000	1.0001	4
5	1.0000|100	1.0000	5

Since the machine is assumed to round-to-even and the printed t corresponds to the $u = 2^{-t}$ used to define x on that iteration we can see that the t value for the last iteration, where $x = \alpha = 1$ after rounding must indicate that there are $t - 1$ bits kept in the mantissa with place values of 2^{-k}.

If round-to-odd or round-up-on-tie was used the table would be

iteration	before rounding	after rounding	t printed
1	1.1000|000	1.1000	1
2	1.0100|000	1.0100	2
3	1.0010|000	1.0010	3
4	1.0001|000	1.0001	4
5	1.0000|100	1.0001	5
6	1.0000|010	1.0000	6

In this case, the knowledge that round-to-odd is used allows us to deduce that the number of bits in the mantissa is $t - 2$ where t is the last printed value.

When a version is executed in MATLAB on `compute1.math.fsu.edu` $t = 53$ and $t = 24$ are observed for a double precision (native MATLAB computation) version and a version that explicitly converts the computed double precision results into single precision in a manner consitent with IEEE rouding.

The results are consistent with those expected in IEEE arithmetic with round-to-even.

The code used is the following.

```
format long

%  double precision part
% assumes native matlab precision is IEEE double

fprintf(1,'Double Precision Loop \n')

x=1.5;
u=1.0;
t=0;
```

```
alpha=1.0;


while ( x > alpha )
  u=u/2;
  x= alpha + u;
  t=t+1;
  fprintf(1,'t = %g u = %g  x = %18.17f alpha = %g \n',t,u,x,alpha)

% note that the t printed here after the increment is the t
% such that the u in x=alpha + u just computed is 2^(-t)
% this is due to the initial division by 2 before the addition

end


%  single precision part
% assumes native matlab precision is IEEE double

fprintf(1,'\n \n Single Precision Loop \n')

x=single(1.5);
u=single(1.0);
t=0;
alpha=single(1.0);

while ( single(x) > single(alpha) )
  u=single(u/2);
  x= single(alpha + u);
  t=t+1;
  fprintf(1,'t = %g u = %g  x = %18.17f alpha = %g \n',t,u,x,alpha)

% note that the t printed here after the increment is the t
% such that the u in x=alpha + u just computed is 2^(-t)
% this is due to the initial division by 2 before the addition

end
```

This yielded the output below with some lines removed for presentation.

```
Double Precision Loop
t = 1 u = 0.5   x = 1.50000000000000000 alpha = 1
t = 2 u = 0.25  x = 1.25000000000000000 alpha = 1
t = 3 u = 0.125   x = 1.12500000000000000 alpha = 1
```

6

```
t = 4 u = 0.0625  x = 1.06250000000000000 alpha = 1
*

*
t = 49 u = 1.77636e-15  x = 1.00000000000000178 alpha = 1
t = 50 u = 8.88178e-16  x = 1.00000000000000089 alpha = 1
t = 51 u = 4.44089e-16  x = 1.00000000000000044 alpha = 1
t = 52 u = 2.22045e-16  x = 1.00000000000000022 alpha = 1
t = 53 u = 1.11022e-16  x = 1.00000000000000000 alpha = 1


 Single Precision Loop
t = 1 u = 0.5  x = 1.50000000000000000 alpha = 1
t = 2 u = 0.25  x = 1.25000000000000000 alpha = 1
t = 3 u = 0.125  x = 1.12500000000000000 alpha = 1
t = 4 u = 0.0625  x = 1.06250000000000000 alpha = 1
*

*

*
t = 20 u = 9.53674e-07  x = 1.00000095367431641 alpha = 1
t = 21 u = 4.76837e-07  x = 1.00000047683715820 alpha = 1
t = 22 u = 2.38419e-07  x = 1.00000023841857910 alpha = 1
t = 23 u = 1.19209e-07  x = 1.00000011920928955 alpha = 1
t = 24 u = 5.96046e-08  x = 1.00000000000000000 alpha = 1
```

Problem 5.4

For this problem assume that the floating point system uses $\beta = 10$ and $t = 3$. The associated floating point arithmetic is such that $x \boxed{op} y = fl(x \ op \ y)$.

Let x and y be two floating point numbers with $x < y$ and consider computing their average $\alpha = (x+y)/2$.

Consider three algorithms for computing α. The parentheses indicate the order of the floating point operations.

- $\alpha_1 = ((x+y)/2.0)$

- $\alpha_2 = ((x/2.0) + (y/2.0))$

- $\alpha_3 = (x + ((y-x)/2.0))$

For the floating point values $x = 5.01$ and $y = 5.02$:

5.4.a. Evaluate α_1, α_2, and α_3 in the specified floating point system.

5.4.b. Explain the results.

5.4.c. Some algorithms produce a series of intervals by splitting an interval (x, y) into intervals (x, α) and (α, y) and choosing to process one of these two smaller intervals further in the next step of the algorithm. Could the behavior observed for the three average computations cause difficulties for such an algorithm?

Solution: The true mean is $\alpha = 5.015$ which correctly rounded is $fl(5.015) = 5.02$. For α_1 we have:

$$fl(x + y) = fl(5.01 + 5.02) = fl(10.03) = 10.0$$
$$\alpha_1 = fl(10.0/2.0) = fl(5.00) = 5.00$$

The computed average is outside the interval! For α_2 we have:

$$fl(x/2.0) = fl(5.01/2.0) = fl(2.505) = 2.50$$
$$fl(y/2.0) = fl(5.02/2.0) = fl(2.510) = 2.51$$
$$\alpha_2 = fl(2.50 + 2.51) = fl(5.01) = 5.01 = x$$

The computed average is one of the endpoints but not the correctly rounded one under round to even. If we round up rather than to even we have

$$fl(x/2.0) = fl(5.01/2.0) = fl(2.505) = 2.51$$
$$fl(y/2.0) = fl(5.02/2.0) = fl(2.510) = 2.51$$
$$\alpha_2 = fl(2.51 + 2.51) = fl(5.02) = 5.02 = y$$

which is the correctly rounded mean. For α_3 we have:

$$fl(y - x) = fl(5.02 - 5.01) = fl(0.01) = 0.01$$
$$fl(0.01/2.0) = fl(0.005) = 0.005$$
$$\alpha_3 = fl(5.01 + 0.005) = fl(5.015) = 5.02 = y$$

This is the true mean correctly rounded for our choice of rounding.

The problem is that the interval has become too small relative to the precision of the arithmetic. The value of α_1 is a disaster for algorithms that bisect intervals as a basic part of their step. The program will fail unless a check is made to ensure the computed mean is within the closed interval $[x, y]$. A carefully designed algorithm will obviate this. Information on the size of the interval was lost when the significant digits in $x + y$ pushed the relevant information out of the set of significant digits kept after rounding.

The value of α_3 is reliable here and indicates that the interval is too small to alter by bisection at this precision. This is due to the fact that it computes the size of the interval and rounds it as part of the computation of the mean.

The value of α_2 stays in the interval and a check would reveal no progress is possible in a interval bisection algorithm. However, we see that the rounding choice that dictates our definition of the true mean rounded and the algorithm choice can cause, in this case, α_2 to be unreliable. So care must be taken to choose the algorithm so that the rounding on the machine and the rounding desired of the true mean are consistent.