Homework 2 Foundations of Computational Math 1 Fall 2017

Problem 2.1

Suppose the n-bit 2's complement representation is used to encode a range of integers, $-2^{n-1} \leq x \leq 2^{n-1} - 1$.

2.1.a. If $x \geq 0$ then $-x$ is represented by bit pattern obtained by complementing all of the bits in the binary encoding of x, adding 1 and ignoring all bits in the result beyond the n-th place, i.e., the bit with weight 2^{n-1}. This procedure is also used when $x < 0$ to recover the encoding of $-x \geq 0$. What is the relationship between the binary encoding of $-2^{n-1} \leq x \leq 2^{n-1} - 1$ and the binary encoding of $-x$ in terms of the number of bits n?

2.1.b. Show that simple addition modulo 2^n on the encoded patterns is identical to integer addition (subtraction) for $-2^{n-1} \leq x,\ y \leq 2^{n-1} - 1$. You may ignore results that are out of range, i.e., overflow.

2.1.c. Show how overflow in addition (subtraction) can be detected efficiently.

2.1.d. Multiplying an unsigned binary number by 2 or 1/2 corresponds to shifting the binary representation left and right respectively (a so-called logical shift). Show how multiplying signed integers encoded via 2's complement representation by 2 or 1/2 can be done via a shifting operation (an arithmetic shift).

Solution:

The range of integers represented is $-2^{n-1} \leq x \leq 2^{n-1} - 1$. If x is an integer in the range of representation and $b(x)$ denotes its encoding in n bits then the two's complement of $b(x)$ encodes $-x$ and the encodings satisfies

$$2^n = b(x) + b(-x)$$

Arithmetic and representations toss out the $n+1$-st bit, the carry-out with weight 2^n, i.e., the arithmetic on encoded words is mod 2^n. The example of $n = 3$ illustrates the pattern of encoding and the relationship between a two's complement pair.

Binary	two's complement
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

For two nonnegative integers $0 \leq x,\ y \leq (2^{n-1}-1)/2 = 2^{n-2} - 2^{-1}$, the sum satisfies $x+y \leq 2^{n-1}-1$ and therefore does not exceed the representation range of n bits for positive numbers. Also, since it is less than 2^n, arithmetic modulo 2^n on the encodings is identical to standard arithmetic, i.e.,

$$x + y = b(x) + b(y) = b(x) + b(y) \bmod 2^n$$

For two negative integers $-2^{n-2} \leq x,\ y < 0$, the sum satisfies $0 > x + y > -2^{n-1}$ and therefore does not exceed the representation range of n bits for negative numbers. For the sum we have

$$\begin{aligned}(b(x) + b(y)) \bmod 2^n &= (2^n - b(|x|) + 2^n - b(|y|)) \bmod 2^n \\ &= [2^n + (2^n - b(|x|) - b(|y|))] \bmod 2^n \\ &= 2^n - b(|x|) - b(|y|) \\ &= b(-(|x| + |y|))\end{aligned}$$

as desired.

To detect overflow note that if x and y are in the representable range and of opposite sign then the result cannot exceed the representable range.

Next consider adding the largest positive representable number to itself using the encoding. We have

$$\begin{aligned}x &= 2^{n-1} - 1 = b(x) \\ b(x) + b(x) \bmod 2^n &= 2^n - 2 < 2^n - 1\end{aligned}$$

and since $2^n - 1$ is the largest codeword we see that the largest result from adding two positive numbers does not wrap around the set of codewords, i.e., the quotient mod 2^n stays 0. Also note that this means that the encoding of the result of any two positive integers that is larger than $2^{n-1} - 1$ must have a 1 in the leftmost bit (the sign bit). Therefore, a simple test for the sum of two positive numbers exceeding the representable positive range is that the sign bit of the result changes to 1 from the 0 in the sign bit of the two operands.

A similar argument for two negative numbers shows that the encoding of the sum creates a quotient mod 2^n of 1 and a sign bit of 0, i.e., it is not possible for the sum to wrap completely around and back to a sign bit of 1 if it exceeds the range of representable negative numbers. The test for overflow is, therefore, only applied when the operands have the same sign and overflow is detected when the sign of the result changes from that of the operands.

This test can also be characterized in terms of the carry-in and carry-out of the leftmost bit in the sum. If the carry-in and carry-out of the n-th bit are different then the sign bit must change in the result and overflow is detected.

An arithmetic shift of an encoding of a positive integer is identical to a logical shift of the codeword. For shifts to the left (doubling the number), a 0 is shifted into the rightmost bit, e.g.,

$$001 \rightarrow 010.$$

Of course, if the sign bit changes from 0 to 1 as the result of the shift then overflow has occurred. For shifts to the right (halving the number), a 0 is shifted into the leftmost bit, e.g.,

$$010 \to 001.$$

The arithmetic shift of a negative number's encoding requires extra consideration for two's complement. For a left shift, as with positive number encodings, a 0 is shifted into the rightmost bit. If the sign bit changes then the range for negative numbers has been exceeded. However, when shifting right to halve a number, rather than shifting in a 0, a 1 is shifted in. For example, doubling -2 to -4 and halving -2 to -1 are

$$110 \to 100$$
$$110 \to 111$$

Therefore for both positive and negative numbers a right shift to halve the number a copy of the sign bit is shifted into the leftmost bit.

Problem 2.2

Consider the following numbers:

- 122.9572
- 457932
- 0.0014973

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

2.2.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.

2.2.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.

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

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 is easily described.

The basic idea is to generate enough of the bits in the potentially nonterminating representation to then apply whatever rounding is desired for the encoding.

Consider 122.9572. In general, if the number is negative then the positive is encoded and the sign attached at the end. The integer portion 122 is easily converted. We start with the largest power of 2 smaller than or equal to 122

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

The fractional part can also be converted to an expansion in terms of 2^{-i} by repeated comparison and subtraction. This can be described as follows

Generates n fractional binary bits into $b(1:n)$
f=0.9572;
s=1.0;
n=40;
b(1:n)=0;
for k=1:n
    s=(s)/(2.0);
    if $f \geq s$
        $b(k) = 1$;
        $f = f - s$;
    end
end

Note the descriptions for the conversion of the integer and fractional parts assume that all of the operations performed are exact.

Using this approach 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 then rounding yields:

$$f = 122.9572$$
$$= 1.11101011110101000010110|0001111001001111 \times 2^6$$
$$\left[\ \sigma \,\middle|\, \epsilon \,\middle|\, \mu\ \right] = \left[\ 0 \,\middle|\, 10000101 \,\middle|\, 11101011110101000010110\ \right]$$

Repeating the exercise on the other two numbers yields:

$$f = 457932 = 1.10111111001100110000000 \times 2^{18}$$
$$\left[\ \sigma \,\middle|\, \epsilon \,\middle|\, \mu\ \right] = \left[\ 0 \,\middle|\, 10010001 \,\middle|\, 10111111001100110000000\ \right]$$
$$f = 0.0014973 = 1.10001000100000100001101000|100001 \times 2^{-10}$$
$$\left[\ \sigma \,\middle|\, \epsilon \,\middle|\, \mu\ \right] = \left[\ 0 \,\middle|\, 01110101 \,\middle|\, 10001000100000100001101001\ \right]$$

Note that the rounding applied to the first and third numbers does not involve a tie so the nearest FP number is simply used.

Of course, implementing this code on a machine that uses IEEE FP arithmetic is not particularly useful since one could simply examine the raw binary form of the number once it was stored in a variable. Nevertheless, it is instructive to consider whether the algorithms above can be done reliably using IEEE FP arithmetic and integer arithmetic.

Since the integer portion of the symbolic specification of the number to be expressed is easily obtained and subtracting powers of 2 from positive integers within range is exact the first portion of the algorithm is reliable.

It is the conversion of the fractional part using the algorithm above that requires a bit of care. The simplest implementation of the algorithm above simply uses double precision representation and compuations in the hope that 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 $f \geq s$
        $b(k) = 1;$
        $f = f - s;$
    end
end

All of the divisions by two are performed exactly and amount to simply shifting the double precision mantissa to the left one bit at a time. The catch lies in the initial assumption that

using double precision for the variable in which f is stored will give all of the correct bits for the subsequent shifts. This is only true in general if chopping is used when f is encoded in double precision. Under certain circumstances if another rounding is used then the double precision version of f when used in the algorithm does not result in the correctly rounded single precision conversion of the fractional part.

Consider a simple example with f entirely fractional where $t = 4$ for single precision and $t = 8$ for double precision. The two vertical lines separate single, double and the rest of the bits.

$$f = .1111|0111|101 \quad \text{true value}$$
$$f_1 = .1111|1000 \quad f \text{ rounded (up) to double}$$
$$f_2 = .1110 \quad f_1 \text{ rounded (to even due to tie) to single}$$

$$f_c = .1111|0111 \quad f \text{ chopped}$$
$$f_s = .1111 \quad f_c \text{ rounded to single}$$

This is also related to the problem of double rounding vs. rounding.

Problem 2.3

2.3.a. Suppose $x \in \mathbb{R}$ and $y \in \mathbb{R}$ with $x < y$. Is it always true that $fl(x) < fl(y)$ in any standard model floating point system?

Solution: This is not always true. Consider a floating point number f such that $x < f < y$. If f is the closest floating point number to x and the closest floating point number to y then for some rounding schemes $fl(x) = f = fl(y)$ and the strict inequality $fl(x) < fl(y)$ does not hold.

2.3.b. Suppose x, y and z are floating point numbers in a standard model floating point arithmetic system. Is floating point arithmetic associative, i.e., is it true that

$$(x \boxed{op} (y \boxed{op} z)) = ((x \boxed{op} y) \boxed{op} z) \ ?$$

Solution: Floating point arithmetic is not associative. To prove it we need a counterexample. Recall our discussion of cancellation. Stewart in his 1998 text gives this now standard example:

$$x = 472635.0000 \ \ y = 27.5013 \ \ z = -472630.0000$$
$$fl(fl(472635.0000 + 27.5013) - 472630) = 33$$
$$fl(27.5013 + fl(472635 - 472630)) = 32.5013$$

In general, all four basic floating point operations are not associative. They are commutative.

2.3.c. Is floating point arithmetic distributive, i.e., is it true that

$$fl(fl(x+z)*y) = fl(fl(fl(x*y)+fl(y*z)))?$$

Solution: Floating point is not in general distributive. It is easy to generate examples that fail because of overflow, e.g., $ab+ac$ has a term that overflows but $a(b+c)$ does not overflow by carefully considering the signs and relative magnitudes of b and c.

Using the same numbers as above and 6 digit decimal arithmetic yields

$$fl(fl(x+z)*y) = 137.507 \quad or \quad 137.506$$
$$fl((fl(xy)+fl(yz))) = 200$$

Problem 2.4

Consider the function

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

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

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

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

2.4.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

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

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

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

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

So the predictions are accurate.

Problem 2.5

Let $f(\xi_1,\ \xi_2, \ldots, \xi_k)$ be a function of k real parameters ξ_i, $1 \le i \le k$. Recall, the relative condition number of f with respect to ξ_1 can be expressed

$$\kappa_{rel} = \max(1, c(\xi_1, \xi_2, \ldots, \xi_k))$$

where $0 \le c(\xi_1, \xi_2, \ldots, \xi_k)$ is a value that indicates the sensitivity of f to small relative perturbations to ξ_1 as a function of the parameters ξ_i, $1 \le i \le k$. If $c(\xi_1, \xi_2, \ldots, \xi_k) \le 1$ then f is considered well-conditioned. Additionally, however, when $c < 1$ its value gives important information. The smaller c is the less sensitive f is to a relative perturbations in ξ_1.

Let $n \ge 2$ be an integer and $\beta > 0$. Consider the polynomial equation

$$p(x) = x^n + x^{n-1} - \beta = 0.$$

2.5.a. Show that the equation has exactly one positive root $\rho(\beta)$.

2.5.b. Derive a formula for $c(\beta, n)$ that indicates the sensitivity of $\rho(\beta)$ to small relative perurturbations to β.

2.5.c. Derive a upper bound on $c(\beta, n)$.

2.5.d. Comment on the conditioning of $\rho(\beta)$ with respect to β.

Solution:

The text by Walter Gautschi, "Numerical Analysis: An Introduction", Birkhuser, Boston, 1997 discusses this problem and related numerical topics. It is recommended as an excellent reference text.

Given $p(x) = x^n + x^{n-1} - \beta$ and letting $\rho(\beta)$ be the positive root of interest we have

$$[\rho(\beta)]^n + [\rho(\beta)]^{n-1} - \beta = 0.$$

Differentiating with respect to β and dropping the argument for convenience yields

$$n\rho^{n-1}\rho' + (n-1)\rho^{n-2}\rho' - 1 = 0$$

$$\begin{aligned}\rho' &= \frac{1}{n\rho^{n-1} + (n-1)\rho^{n-2}} = \frac{\rho}{n\rho^n + (n-1)\rho^{n-1}} \\ &= \frac{\rho}{n\rho^n + (n-1)(\beta - \rho^n)} = \frac{\rho}{(n-1)\beta + \rho^n}\end{aligned}$$

$$c(\beta) = |\frac{\beta\rho'}{\rho}| = \frac{\beta}{(n-1)\beta + \rho^n} = \frac{1}{(n-1) + \frac{\rho^n}{\beta}}$$

Since $\rho > 0$ and $\beta > 0$ we have

$$c(\beta) < \frac{1}{n-1}$$

indicating a well-conditioned root for $n \geq 2$ that gets less sensitive as n increases.

Problem 2.6

The evaluation of

$$f(x) = x\left(\sqrt{x+1} - \sqrt{x}\right)$$

encounters cancellation for $x \gg 0$.

Rewrite the formula for $f(x)$ to give an algorithm for its evaluation that avoids cancellation.

Solution:

$$\begin{aligned}f(x) &= x\left(\sqrt{x+1} - \sqrt{x}\right) \\ &= x\left(\sqrt{x+1} - \sqrt{x}\right) \ \frac{\left(\sqrt{x+1} + \sqrt{x}\right)}{\left(\sqrt{x+1} + \sqrt{x}\right)} \\ &= \frac{x}{\left(\sqrt{x+1} + \sqrt{x}\right)}\end{aligned}$$

No cancellation occurs when turning this expression into an algorithm.

Problem 2.7

2.7.a

Suppose that x and y are two floating point numbers in a system that supports gradual underflow and satisfies the standard model. Show that if $y/2 \leq x \leq 2y$ then

$$fl(x - y) = x - y$$

2.7.b

Suppose a triangle has sides with lengths $a \geq b \geq c$. Heron's formula for its area is

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2}$$

Kahan has suggested the following formula

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}$$

(i) What happens with the Heron's formula with needle-shaped triangles?

(ii) Give an informal proof that Kahan's formula is reliable numerically. You may consult the literature of course.

(iii) Compare the accuracy of the two formulae in single-precision for several examples to illustrate your points.

Solution:

The results on exact subtract are discussed in Sterbenz 1974 text *Floating point computations*, Prentice Hall and in Chapter 2 of Higham's, *Accuracy and stability of numerical algorithms*, SIAM, Second Edition. The former is in the context of pre-IEEE standard floating point systems. The latter also discusses a more general result due to Ferguson.

The results are for any base but we restrict our discussions to $\beta = 2$. The assumption $y/2 \leq x \leq 2y$ implies that x and y are both positive so the problem is one of considering when subtraction with close magnitudes is exact.

Assume further that x and y normalized floating point numbers with exponents e_x and e_y respectively. The assumption implies that one of three situations: $e_x = e_y$, $e_x = e_y + 1$, and $e_x = e_y - 1$. We also assume for the moment that the exponents are large enough so that no difference $x - y$ is a subnormal floating point number.

If $e_x = e_y$ then no shift is needed to align exponents and the subtraction is that of two t bit mantissas with a result mantissa that must be exactly representable in t bits. Any left shift to impose normalization does not cause a loss of digits and the exact result is maintained.

If $e_x = e_y + 1$ then a right shift of exactly 1 is needed to align exponents. This implies that the mantissa of x is $1\ b_{t-1}\ \ldots\ b_1\ |0$ where the 0 to the right of the vertical line is a guard bit of which we assume at least one. The mantissa for the shifted y is $0\ 1\ \tilde{b}_{t-1}\ \ldots\ \tilde{b}_2\ |\tilde{b}_1$ where the leading 0 has been introduced by the shift and the rightmost bit has been moved into the guard bit position. The assumption $x \leq 2y$ implies that the difference $x - y \leq y$ and therefore the mantissa of $x - y$ has at least one leading 0. This implies a left shift of at least one to normalize the result and the leftmost guard bit is returned to the normalized mantissa and the result is exact.

If $e_x = e_y - 1$ the assumption that $x \geq y/2$ and a similar argument shows the exact subtraction desired.

The only situation not covered is when the exponents are small enough to result in a subnormalized difference. The assumption is that the system supports gradual underflow, i.e., subnormal numbers are part of the floating point number system. In these cases, an exact subtraction follows once again.

The accuracy of Kahan's alternative for the area of a triangle is stated formally in Theorem 3 of Goldberg's paper posted on the class webpage and it is discused in Chapter 2 of Higham's text. See also the classic manuscript of Kahan at www.cs.berkeley.edu/ wkahan/MathSand.pdf.

A needle-shaped triangle has two long sides $a > b$ and a relatively very short side c so that $a \approx b + c$. In Heron's formula then $s \approx a$ and $s - a$ will suffer significant cancellation in

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2}$$

Kahan's alternative

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}$$

is more reliable. Higham gives the following informal argument based on the result in the first part of the problem.

The sum $a + (b + c)$ is a sum of positive numbers and is therefore computed accurately. The sums $c + (a - b)$ and $a + (b - c)$ are the sums of two positive numbers and therefore computed accurately. The problem and assumptions imply $a \leq b + c$ and

$$b \leq a \leq b + c \leq 2b$$

and therefore $a - b$ is computed exactly. The difference $c - (a - b)$ is the difference of two exactly represented numbers and is therefore computed accurately. Given that the argument to the square root is therefore accurately evaluated and the square root is a basic function assumed to satisfy the standard model, the result is accurate.