CSC420 Assignment 1

Monica Li
Student Number: 997440899
CDF: gwicked

October 8, 2015

1. (a) In IEEE arithmetic how many double precision numbers are there between any two adjacent nonzero single precision numbers? Justify your response.

The IEEE standard gives us the following values for single and double precision: Single precision: $\beta = 2$, t = 24, L = -126, U = 127 Double precision: $\beta = 2$, t = 53, L = -1022, U = 1023

For binary digits, you have only two choices (0 or 1). The difference in digits between single and double precision is $53 - 24 = 29$. Therefore there are 2^{29} double precision numbers between any two adjacent single precision numbers.

(b) What is the largest integer p such that all integers in the interval $[-p, p]$ are exactly representable in IEEE double precision arithmetic? What is the corresponding p for IEEE single precision arithmetic? Justify your responses.

The largest integer p such that all integers in the interval $[-p, p]$ are exactly representable in IEEE floating point arithmetic can be calculated as follows:

$$p = 2^t \text{ for } [-p, p]$$

So, the largest integer p for for IEEE double precision arithmetic is $p = 2^{53}$, approximately $9.01x10^{15}$. The largest integer p for for IEEE single precision arithmetic is $p = 2^{24}$, approximately $1.68x10^{7}$.

2. (a) The following C program:

```
#include <stdio.h>

int main (void) {
    double a, b;
    for (a = 0.1; a <= 0.5; a += 0.1)
        printf("a = %e\n", a);
    for (b = 1.1; b <= 1.5; b += 0.1)
        printf("b = %e\n", b);
return 0;
}
```

outputs five values of `a` and four values of `b`. Explain why the two loops do not produce the same number of lines of output.

(Ask for assistance if you do not know C. The programming language is not the issue here.)

They do not produce the same number of lines of output because $a = 0.5$ is exactly representable. When calculating b to 1.5, we find that the value stored is not quite 1.5, and is in fact slightly greater than 1.5, breaking the loop condition.

(b) Suppose that you are writing a program to run on a computer that supports IEEE floating point arithmetic, have declared and initialize a floating point variable named `x`, and need to determine 10% of the value stored in `x`. You need to decide between using the expression `0.1 * x` or `x / 10.0`. Which of these two expressions would you recommend using if accuracy of the final result is your biggest concern? Explain.

Although these particular numbers produce very similar results, I would choose the latter $x/10.0$ as the more accurate expression. The number 10.0 is exactly representable when stored as a double, whereas 0.1 is not. Multiplication and division can produce the about the same amount of error, but the conversion of 0.1 may cause rounding errors, which makes the result of $0.1 * x$ less accurate.

3. You know that cos($2m\pi$) = 1 for every integer m. Using a programming language of your choice that uses IEEE fixed precision floating-point arithmetic, write a program that calculates `cos( 2*m*pi )` for `m` = $10.0^{\texttt{k}}$, `k` = $0, 1, 2, \ldots, 20$, where `pi` is either your language's builtin approximation to π or `4.0*arctan(1.0)`. Present the results of your calculations in a formatted table, with column headings for the values of `k`, `2*m*pi`, and `cos( 2*m*pi )`. Use a "scientific notation" to present the values of `2*m*pi` and `cos( 2*m*pi )`, giving 15 significant figures for both `2*m*pi` and `cos( 2*m*pi )`.

Carefully explain why you do not always compute `cos(2*m*pi)` ≈ 1.0 and why the approximations change significantly for larger values of `k`.

Listing 1: Code

```
#include <stdio.h>
#include <math.h>

int main(void) {
        float m;
        int k;

        // Header
        printf("k\tm\tcos(2*m*pi)\t2*m*pi");
        // Loop through values
        for(k = 0; k <= 20; k++) {
                m = pow(10, k);
                printf("\n%d\t%.15E\t%.15E\t%.15E", k, m, cos(2*m*M_PI),
                    2*m*M_PI);
        }
return 0;
}
```

The calculation for $m = 10.0^k$ has small error margins, which don't affect the value much for small values of k. These small changes result in small errors in the result. Storing larger values as a floating point becomes inaccurate. The proportion of relative error remains small, but as k values increase, the absolute error increases and compound upon each other (especially as cosine is a repeating function), directly affecting the final output of cosine.

Table 1: Values Calculated for $\cos(2m\pi)$ and $2m\pi$

k	m	cos(2*m*pi)	2*m*pi
0	1.000000000000000E+00	1.000000000000000E+00	6.283185307179586E+00
1	1.000000000000000E+01	1.000000000000000E+00	6.283185307179586E+01
2	1.000000000000000E+02	1.000000000000000E+00	6.283185307179587E+02
3	1.000000000000000E+03	1.000000000000000E+00	6.283185307179586E+03
4	1.000000000000000E+04	1.000000000000000E+00	6.283185307179586E+04
5	1.000000000000000E+05	1.000000000000000E+00	6.283185307179586E+05
6	1.000000000000000E+06	1.000000000000000E+00	6.283185307179586E+06
7	1.000000000000000E+07	1.000000000000000E+00	6.283185307179587E+07
8	1.000000000000000E+08	9.999999999999969E-01	6.283185307179586E+08
9	1.000000000000000E+09	9.999999999999978E-01	6.283185307179586E+09
10	1.000000000000000E+10	9.999999999899705E-01	6.283185307179586E+10
11	9.999999795200000E+10	9.999999997709935E-01	6.283185178499951E+11
12	9.999999959040000E+11	9.999999610710738E-01	6.283185281443659E+12
13	9.999999827968000E+12	9.999999497446436E-01	6.283185199088693E+13
14	1.000000003768320E+14	9.982970936167014E-01	6.283185330856639E+14
15	9.999999869911040E+14	9.645408935198392E-01	6.283185225442282E+15
16	1.000000027256422E+16	9.640747464739566E-01	6.283185478436739E+16
17	9.999999843067494E+16	8.133345243286308E-01	6.283185208575985E+17
18	9.999999843067494E+17	9.972758033389917E-01	6.283185208575985E+18
19	9.999999980506448E+18	8.935424534002419E-01	6.283185294931427E+19
20	1.000000020040877E+20	-8.014931780156964E-01	6.283185433100132E+20

4. Consider the problem of evaluating the function $f(x)$ for any $x \in [-\pi/2, +\pi/2]$, where

$$f(x) = \begin{cases} \dfrac{1-\cos(x)}{\sin(x)}, & x \in [-\pi/2, 0) \cup (0, +\pi/2] \\ 0, & x = 0. \end{cases}$$

(a) Show why one can expect to be able to write a computer program that uses floating-point arithmetic to accurately evaluate $f(x)$ at any $x \in [-\pi/2, +\pi/2]$.

We can find out whether we can expect to write a computer program that uses floating-point arithmetic accurately by looking at the condition number of the function. The condition number can be calculated by the following formula:

$$\begin{aligned} \kappa(f(\bar{x})) &= |\frac{\bar{x}f'(\bar{x})}{f(\bar{x})}| \\ \kappa(\frac{1-\cos(x)}{\sin(x)}) &= |\frac{\bar{x}f'(\frac{1-\cos(x)}{\sin(x)})}{\frac{1-\cos(x)}{\sin(x)}}| \\ &= |\frac{\bar{x}(\csc(x)(\csc(x) - cot(x)))}{\frac{1-\cos(x)}{\sin(x)}}| \\ &= |\frac{\bar{x}(\csc(x) - cot(x))}{1-\cos(x)} \cdot \cancel{\csc(x)\sin(x)}| \\ &= |\frac{\bar{x}(\frac{\cancel{1-\cos(x)}}{\sin(x)})}{\cancel{1-\cos(x)}}| \\ &= \frac{x}{\sin(x)} \end{aligned} \tag{1}$$

We see in (1) that the condition number is $\frac{x}{\sin(x)}$. For the interval given above, the values of the condition number are relatively small, reaching 1.6 at the bounds. Since the condition number is bounded and not arbitrarily high, the function is not too ill-conditioned and can be written in a way that can be accurately evaluated. We can avoid catastrophic cancellation by converting the original equation to another, more stable equation that does not include the error caused by substraction (see part (b)).

(b) Derive an accurate algorithm for evaluating $f(x)$ and explain why you think that accurate results will be computed. Make use of standard math library functions like `sin()` and `cos()` when expressing your algorithm. We can create a more accurate algorithm by getting rid of operations that cause large errors, such as subtraction.

Let's start by trying to convert by substituting $y = \frac{x}{2}$ and using trig identities to convert the original equation to one without subtraction

$$\begin{aligned} f(y) &= \frac{1-\cos(2y)}{\sin(2y)}, y = \frac{x}{2} \\ &= \frac{1-(1-2\sin^2(y))}{2\sin(y)\cos(y)} \\ &= \frac{\cancel{2}\sin^{\cancel{1}}(y)}{\cancel{2\sin(y)}\cos(y)} \\ &= \frac{\sin(y)}{\cos(y)} \\ &= \tan(y) \end{aligned} \tag{2}$$

We substitute x back into the equation and can see the new equation.

$$f(x) = \frac{1-\cos(x)}{\sin(x)} = \tan(\frac{x}{2})$$

This equation is much more stable than the original, as we avoid catastrophic cancellation.

Listing 2: Algorithm to solve f(x) more accurately

```
#include <stdio.h>
#include <math.h>

int main(void) {
    double i, v;
    for(i = -M_PI/2; i <= M_PI/2; i += M_PI/2) {
        v = tan(i/2);
        printf("\nThe value of f(x) at %.10f is %.10f", i, v);
    }
return 0;
}
```

Note: The for loop is simply to print out a couple values of i: $-\frac{\pi}{2}, 0, \frac{\pi}{2}$, and not part of the algorithm to calculate f(x). The algorithm to calculate is simply to divide your input by 2 and get tan of that value.