

**Assignment 2 - Floating Point in C, Over/Underflow,
Numerical Cancellation**

COMP 350 - Numerical Computing
Prof. Chang Xiao-Wen
Fall 2018

LE, Nhat Hung

McGill ID: 260793376
Date: September 24, 2018
Due date: September 27, 2018

1. (5 points) Write a C program to find the smallest positive integer x such that the floating point expression

$$1 \oslash (1 \oslash x)$$

is not equal to x , using single precision. Make sure that the variable x has type float, and assign the value of the expression $1 \oslash x$ to a float variable before doing the other division operation. Repeat with double precision.

```
void question1Single() {
    float x, invOfX;
    x      = 1.0;
    invOfX = 1.0 / x;

    while (x == 1.0 / invOfX) {
        x++;
        invOfX = 1.0 / x;
    }

    printf("Question 1 single precision: %e\n", x);
}

void question1Double() {
    double x, invOfX;
    x      = 1.0;
    invOfX = 1.0 / x;

    while (x == 1.0 / invOfX) {
        x++;
        invOfX = 1.0 / x;
    }

    printf("Question 1 double precision: %e\n", x);
}
```

In single precision, the smallest integer x satisfying the above is 3.0.
In double precision, 49.0.

2. (5 points) A calculus student was asked to determine $\lim_{n \rightarrow \infty} x_n$, where $x_n = (100^n)/n!$. He wrote a C program in single precision to evaluate x_n by using

$$x_1 = 100, x_n = 100x_{n-1}/n, n = 2, 3, \dots, 70.$$

The numbers printed became ever larger and finally became ∞ . So the student concluded that $\lim_{n \rightarrow \infty} (100^n)/n! = \infty$. Please write a C program in single precision to verify the student's observation. The student's conclusion is actually wrong. What is the problem with his program?

```
void question2() {
    float x_n;
    int    n;

    x_n = 100;
    n    = 2;

    for (; n <= 70; n++) {
        x_n = (100 * x_n) / n;
        printf("n = %d, x_n = %e\n", n, x_n);
    }
}
```

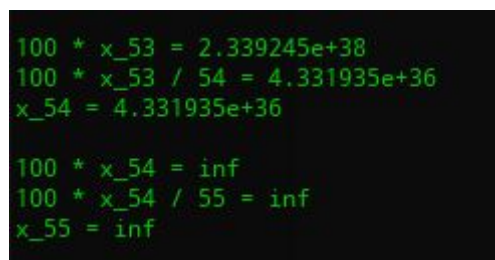
For $n \geq 55$, the program gives ∞ as the result. This explains the student's observation.

The student's conclusion is wrong, however, because the ∞ result is due to an overflow.

The calculation of x_n can be broken down to

$$(100 \otimes x_{n-1}) \oslash n$$

The overflow occurs in $(100 \otimes x_{n-1})$, at $n = 55$:



```
100 * x_53 = 2.339245e+38
100 * x_53 / 54 = 4.331935e+36
x_54 = 4.331935e+36

100 * x_54 = inf
100 * x_54 / 55 = inf
x_55 = inf
```

Figure 1: Inspecting $100 \otimes x_{n-1}$, then $(100 \otimes x_{n-1}) \oslash n$

Indeed, $100 \otimes x_{54} = \infty$. And $\infty \oslash n = \infty$.

In conclusion, the reason the program shows ∞ as the result is because of an overflow of $100 \otimes x_{n-1}$.

Bonus (2 points): Can you rewrite a C program to evaluate x_n so that you can make a right conclusion about $\lim_{n \rightarrow \infty} x_n$?

3. (10 points) In 250 B.C.E., the Greek mathematician Archimedes estimated the number π as follows. He looked at a circle with diameter 1, hence circumference π . Inside the circle he inscribed a square. The perimeter of the square is smaller than the circumference of the circle, and so it is a lower bound for π . Archimedes then considered an inscribed octagon, 16-gon, etc., each time doubling the number of sides of the inscribed polygon, and producing ever better estimates for π . Using 96-sided inscribed and circumscribed polygons, he was able to show that $223/71 < \pi < 22/7$. There is a recursive formula for these estimates. Let p_n be the perimeter of the inscribed polygon with 2^n sides. Then p_2 is the perimeter of the inscribed square, $p_2 = 2\sqrt{2}$. In general

$$p_{n+1} = 2^n \sqrt{2(1 - \sqrt{1 - (p_n/2^n)^2})}.$$

(a) Write a program to compute p_n for $n = 3, 4, \dots, 35$ in **double precision** by using the formula. Explain your results.

```
void question3a() {
    double p_n, p_nMinus1, nMinus1;

    nMinus1 = 2.0;
    p_nMinus1 = 2.0 * sqrt(2.0);

    for (int i = 3; i <= 35; i++) {
        p_n =
            pow(2.0, nMinus1) *
            sqrt(
                2.0 * (1.0 - sqrt(
                    1.0 - pow(p_nMinus1 / pow(2.0, nMinus1), 2.0)
                ))
            );

        printf("p_%d = %e\n", i, p_nMinus1);
        nMinus1++;
        p_nMinus1 = p_n;
    }
}
```

The results are:

```
p_3 = 3.061467e+00    p_14 = 3.141593e+00    p_25 = 3.142451e+00
p_4 = 3.121445e+00    p_15 = 3.141593e+00    p_26 = 3.162278e+00
p_5 = 3.136548e+00    p_16 = 3.141593e+00    p_27 = 3.162278e+00
p_6 = 3.140331e+00    p_17 = 3.141593e+00    p_28 = 3.464102e+00
p_7 = 3.141277e+00    p_18 = 3.141593e+00    p_29 = 4.000000e+00
p_8 = 3.141514e+00    p_19 = 3.141594e+00    p_30 = 0.000000e+00
p_9 = 3.141573e+00    p_20 = 3.141597e+00    p_31 = 0.000000e+00
p_10 = 3.141588e+00   p_21 = 3.141597e+00    p_32 = 0.000000e+00
p_11 = 3.141591e+00   p_22 = 3.141674e+00    p_33 = 0.000000e+00
p_12 = 3.141592e+00   p_23 = 3.141830e+00    p_34 = 0.000000e+00
p_13 = 3.141593e+00   p_24 = 3.142451e+00    p_35 = 0.000000e+00
```

We can see $p_n = 0$ with $n \geq 30$:

This is a result of **numerical cancellation** at $n = 30$ in this part of the formula:

$$1 - (p_{n-1}/2^{n-1})^2$$

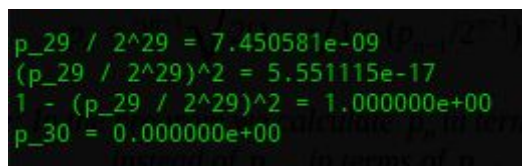
in

$$p_n = 2^{n-1} \sqrt{2(1 - \sqrt{1 - (p_{n-1}/2^{n-1})^2})}$$

*Note: In the program we calculate p_n in terms of p_{n-1} ,
instead of p_{n+1} in terms of p_n .*

At $n = 30$, $(p_{n-1}/2^{n-1})^2$ is so small that $1 - (p_{n-1}/2^{n-1})^2 = 1$.

After that, $1 - \sqrt{1} = 0$, which causes a 0 to appear in the formula, nullifying p_n .



```
p_29 / 2^29 = 7.450581e-09
(p_29 / 2^29)^2 = 5.551115e-17
1 - (p_29 / 2^29)^2 = 1.000000e+00
p_30 = 0.000000e+00
```

Figure 2: $1 - (p_{29}/2^{29})^2 = 1$

We can also see that with $13 \leq n \leq 29$, p_n exceeds the actual value of π (3.141592...):

This is also due to $(p_{n-1}/2^{n-1})^2$ being so small that $1 - (p_{n-1}/2^{n-1})^2$ does not give the real value, but not small enough to make $1 - (p_{n-1}/2^{n-1})^2 = 1$. It is precision loss due to the limit of the fraction field width.

(b) Improve the formula to avoid the difficulty with it. Compute p_n for $n = 3, 4, \dots, 35$ by your new formula in double precision. Comment on your results.

We will now calculate the perimeters of both the inscribed and circumscribed polygons of the circle. The approximation of π , p_n , will then be the average of the two polygons' perimeters.

```
void question3b() {
    double p_n, circumscribed_n, inscribed_n;

    inscribed_n = 2.0 * sqrt(2.0);
    circumscribed_n = 4.0;
```

```

for (int n = 3; n <= 35; n++) {
    circumscribed_n = 2.0 * circumscribed_n * inscribed_n /
        (circumscribed_n + inscribed_n);

    inscribed_n = sqrt(circumscribed_n * inscribed_n);

    p_n = (circumscribed_n + inscribed_n) / 2;
    printf("p_%d = %20.15e\n", n, p_n);
}
}

```

We obtain the following results:

```

p_3 = 3.187587978952740e+00    p_20 = 3.141592653592142e+00
p_4 = 3.152021515166290e+00    p_21 = 3.141592653590379e+00
p_5 = 3.144136698987598e+00    p_22 = 3.141592653589939e+00
p_6 = 3.142224771100329e+00    p_23 = 3.141592653589829e+00
p_7 = 3.141750440437615e+00    p_24 = 3.141592653589802e+00
p_8 = 3.141632085156635e+00    p_25 = 3.141592653589795e+00
p_9 = 3.141602510535137e+00    p_26 = 3.141592653589792e+00
p_10 = 3.141595117766985e+00   p_27 = 3.141592653589793e+00
p_11 = 3.141593269630395e+00   p_28 = 3.141592653589793e+00
p_12 = 3.141592807599713e+00   p_29 = 3.141592653589793e+00
p_13 = 3.141592692092259e+00   p_30 = 3.141592653589793e+00
p_14 = 3.141592663215409e+00   p_31 = 3.141592653589793e+00
p_15 = 3.141592655996197e+00   p_32 = 3.141592653589793e+00
p_16 = 3.141592654191394e+00   p_33 = 3.141592653589793e+00
p_17 = 3.141592653740194e+00   p_34 = 3.141592653589793e+00
p_18 = 3.141592653627393e+00   p_35 = 3.141592653589793e+00
p_19 = 3.141592653599193e+00

```

This algorithm does not involve subtraction, and numbers in operations are neither too small nor too large. There is therefore no cancellation, overflow or underflow. For this reason the results have better precision.