CSC338. Homework 2

Due Date: Wendesday January 21, 9pm

Please see the guidelines at https://www.cs.toronto.edu/~lczhang/338/homework.html

What to Hand In

Please hand in 2 files:

- Python File containing all your code, named `hw2.py`.
- PDF file named `hw2_written.pdf` containing your solutions to the written parts of the assignment. Your solution can be hand-written, but must be legible. Graders may deduct marks for illegible or poorly presented solutions.

If you are using Jupyter Notebook to complete the work, your notebook can be exported as a .py file (File -> Download As -> Python). Your code will be auto-graded using Python 3.6, so please make sure that your code runs. There will be a 20% penalty if you need a remark due to small issues that renders your code untestable.

Make sure to remove or comment out all matplotlib or other expensive code before submitting your code!

Submit the assignment on **MarkUs** by 9pm on the due date. See the syllabus for the course policy regarding late assignments. All assignments must be done individually.

```
import math
import numpy as np
```

Question 1

We'll be implementing a floating point system. For most of this question, assume that we are working with:

- base $\beta = 3$,
- precision $p = 5$, and
- exponent range $[L, U]$ = [-3, 3]

Our floating point system will be normalized.

```
# These are the constants we'll use in our code
BASE = 3
PRECISION = 5
L = -3
U = +3

# We will represent a floating number as a tuple (mantissa, exponent, sign)
# With: mantissa -- a list of integers of length PRECISION
#       exponent -- an integer between L and U (inclusive)
#       sign     -- either 1 or -1
example_float = ([1, 0, 0, 1, 0], 1, 1)
```

Part (a) – 1pt

How many normalized-point numbers are in our system? Use a formula in terms of `BASE`, `PRECISION`, `L` and `U` to compute the count, and save the result in the value `total_num_floats`.

```
total_num_floats = None
```

Part (b) – 3pt

Write a function `is_valid_float` that checks whether a tuple of the form `(mantissa, exponent, sign)` represents a valid, normalized float in our floating point system.

```
def is_valid_float(float_value):
    """Returns a boolean representing whether the float_value is a valid,
    normalized float in our floating point system.

    >>> is_valid_float(example_float)
    True
    """

    (mantissa, exponent, sign) = float_value
    return None
```

Part (c) – 3pt

Construct a floating-point representation tuple of the form `(mantissa, exponent, sign)` of each of the following:

1. The largest negative number representable in our floating point system. Store it in the variable `largest_negative`.
2. The smallest positive number (greater than 0) in our floating point system. Store it in the variable `smallest_positive`
3. The value of 32 represented in our floating system. Assume chopping is used for rounding. Store it in the variable `float_32`.

```
largest_negative = ([], None, None)
smallest_positive = ([], None, None)
float_32 = ([], None, None)
```

Part (d) – 5pt

Write a function `to_num` that converts a tuple of the form `(mantissa, exponent, sign)` to a Python numerical value.

```
def to_num(float_value):
    """Return a Python floating point representation of `float_val`
    These examples are for your understanding, and your actual output
    might vary slightly.

    >>> to_num(example_float)
    3.111111111111111
    """
    (mantissa, exponent, sign) = float_value
    return None
```

Part (e) – 5pt

Write a function `add` that takes two tuples of the form `(mantissa, exponent, sign)`, and performs floating-point addition to obtain a third floating-point number in our representation `(mantissa, exponent, sign)`.

You may assume that the signs of both addends are positive.

```
def add_float(float1, float2):
    """Return a valid floating-point representation of the form (mantissa, exponent, sign)
    that is the sum of `float1` and `float2`. Raises a ValueError if the result of
    the addition is not a valid float.
    >>> add_float(example_float, example_float)
    ([2, 0, 0, 2, 0], 1, 1])
    """
    (mantissa1, exponent1, sign1) = float1
    (mantissa2, exponent2, sign2) = float2

    # You may assume that sign1 and sign2 are positive
    assert (sign1 == 1) and (sign2 == 1)

    # Add your code here
    return (None, None, None)
```

Part (f) – 2pt

Show that `add_float(x,y)` is not associative. Include this part of your work in your PDF file.

Question 2

We are going to show catestrophic cancellation by computing $\frac{1}{1-x}$ and e^x using their taylor series expansion. Recall that:

$\frac{1}{1-x} = 1 + x + x^2 + x^3 + ... = \sum_{n=0}^{\infty} x^n$ for $x \in (-1, 1)$, and

$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

The functions `h1` and `h2` returns a list of the first n elements of the Taylor series expansions of $\frac{1}{1-x}$ and e^x, respectively.

```
def h1(x, n):
    """Returns a list of the first n terms of the Taylor Series expansion of 1/(1-x)."""
    return [pow(x,i) for i in range(n)]

def h2(x, n):
    """Returns a list of the first n terms of the Taylor Series expansion of e^x."""
    return [pow(x,i)/math.factorial(i) for i in range(n)]
```

Part (a) – 3pt

Does computing $\frac{1}{1-x}$ using `sum(h1(x,n))` suffer from catestrophic cancellation? If so, show an example: choose an x where catestrophic cancellation occurs and show the list $h1(x, n)$. If not, briefly explain why not.

Include your solution in the PDF writeup.

Part (b) – 1pt

We already showed in class that computing e^x in this way gives disasterous results for x < 0. For the rest of this part of the assignment, set $x = -30$.

First, compute e^x using `math.exp`.

Now, compute the estimate: `h2(-30, n)` for `n = 20, 40, 60, 80, 100, 120, 140, 160`.

Save the result as a list, in order of ascending values of n, in the variable `exp_estimates`.

```
ns = [20, 40, 60, 80, 100, 120, 140, 160]
exp_estimates = []
```

Part (c) – 2pt

Describe the values of `exp_estimates`. What does the first 4 values look like, and why? Do the estimates eventually converge? Why or why not?

Include your solution in the PDF writeup.

Question 3

Consider the function $z(n)$ below:

```
def z(n):
    a = pow(2.0, n) + 10.0
    b = (pow(2.0, n) + 5.0) + 5.0
    return a - b
```

Part (a) – 1pt

Using Python, find all positive integers `n` for which the value of `z(n)` is nonzero. Save the result in a list called `nonzero_zn`.

```
nonzero_zn = []
```

Part (b) – 4pt

Using what we learned about floating-point addition and the rounding rules, explain why the `z(n)` takes on these particular non-zero values. Why is the expression zero for all other values of n? You may assume that Python uses IEEE Double Precision for floating-point arithmetic (i.e., round to even in base 2 with a mantissa of 53 bits). Hint: look at the rounding error produced by each of the three additions.