Etude 4
Isabel Baltazar 8093074 Lencho Burka 1038530 Nikhil Patil 75669

All of the following tests have rounding and truncation errors caused by floating point arithmetic. Floating point numbers can be 32 bits long for doubles or 64 bits long for floats. Floating point numbers are only approximation, so a number like 0.1 is actually stored as 0.10000000000000003. This is because converting a decimal number like 0.1 to binary results in a repeating number 0.00011001100…, similar to the way ⅓ becomes 0.3333… in a normal math problem. Storing 0.1 in binary would require an infinite amount of memory, but since the computer is limited to 32 bits, we have to round up, giving 0.10000000000000003. In summary, not all decimal numbers can accurately represented in binary.

Harmonic Numbers

1)

Test 1:
Sum of single precision harmonic numbers where n = 100000:
forward: 9.787613
backward: 9.787604
The answers agree up to the 4th decimal place.

Test 2:
Sum of double precision harmonic numbers where n = 100000:
forward: 9.787606036044348
backward: 9.787606036044386

The answers agree up to the 13th decimal place.

Precision is defined by how large the difference between the computer's approximation of a number and the actual value. For example, computers store 0.1 as 0.10000000000000003, meaning there is a 3e10-15 difference between the computer's approximation and 0.1.

2)

Double precision numbers are 64 bits long, while single precision numbers are only 32 bits long. This means that for very long numbers, single precision numbers will lose a lot of accuracy thanks to rounding/truncation errors. This is demonstrated by the results of Test 1, where the single precision numbers only agree up to 4 decimal places while in Test 2, the results agree up to the 13th decimal place.

The backward calculation of harmonic numbers is more accurate because we start with a number with a lot of decimal places (1/10000). If we used the forward calculation, we would start with a slightly smaller number, and lose precision when we get to 1/10000 because it has to be rounded up to fit the smaller number.

Etude 4
Isabel Baltazar 8093074 Lencho Burka 1038530 Nikhil Patil 75669

We tested our hypothesis using f(n), where the result of f(n) should be approximately equal to n. We used three n-values {0, 1000, 10000}

$$f(n) = 1 - 1 + 1 + ½ + … 1 - 1/n$$

For n = {0, 1000}, our hypothesis was correct. The backward calculation of harmonic numbers were resulted in a closer approximation of n.

n = 0
Forward Test:
Accuracy: 0.0 - 0 = 0.0
Backward Test:
Accuracy: 0.0 - 0 = 0.0
n = 1000
Forward Test:
Accuracy: 992.51434 - 1000 = |7.4856567|
Backward Test:
Accuracy: 992.5147 - 1000 = |7.4852905|

However, when n = 10,000, we found that the forward calculation was a closer approximation of n. So for larger n values, we found that truncation errors are more noticeable in the backwards calculation. Our hypothesis is wrong for large n values.

n = 10,000
Forward Test:
Accuracy: 9991.131 - 10000 = |8.869141|
Backward Test:
Accuracy: 9989.909 - 10000 = |10.09082|

Etude 4
Isabel Baltazar 8093074 Lencho Burka 1038530 Nikhil Patil 75669

Standard Deviation:

Results of StandardDeviation.java

Results for Array: [3 5 2 7 6 4 9 1]
Stdv Method 1: 2.496873044429772
Stdv Method 2: 2.496873044429772

100 added to each element.
Method 1: 2.496873044429772
Method 2: 2.496873044429772

Random array of length 10,000
Stdv Method 1: 2.5826454576654543
Stdv Method 2: 2.5826454576654543

100 added to each element.
Method 1: 2.5826454576654543
Method 2: 2.582645457665592

1) Method 1 and 2 agree for small sets of numbers.
When we increase the array length to 10,000, the two methods start differing at around the 7th decimal place.
res

This is because every floating point operation introduces some error to the calculation, and as we increase the number of operations, so does the size of the error.

2) Method 1 gives the same result if you increase each element in the set by the same amount. Method 2's results slightly differ because of how the computer handles float inconsistencies and rounding.

Every floating point arithmetic operation introduces some error to the result, and Method 2 has one more division operation than Method 1.

We observed that upon adding a fixed value of > 1 000 000 000(1 Billion) and at times > 100 000 000(One hundred Million), The standard deviation of method 1 remains is not changed but method 2 crashes producing values such as 0., NaN, values significantly different to each other. 1 Billion is the maximum significant figure doubles can store, therefore since method 2 does not have a pre-computated value of the mean the numerator before the first division occurs, is greater than the maximum number of significant figures doubles has memory to store resulting in a an overflow.

Isabel Baltazar 8093074 Lencho Burka 1038530 Nikhil Patil 75669

For an array containing ten copies of 0.0, Method 1 and Method 2 work as normal. However, with 15 copies of 0.001, Method 1 and Method 2 both crash because of the machines has trouble approximating very small floating point numbers.

3) Method 1 is preferred because it avoids the truncation errors of Method 2. Method 1 is more stable even for large sets of numbers. Method 2 is preferred if you have limited memory because method 1 requires more space to store the pre-computated values.

However, I would not use either method to count money. We need all of those decimal places to be accurate.

Identities of Real Numbers:

```
f(y) = ((x/y) - x(y)) * y + x(y)(y)
```

3) Is x = f(y) always true when 1 < y < 100?

The formula failed for these y values when using floats. We allowed x and x(n) to have a difference of up to 1E-10.

x	y	Difference	Difference < 1E-10?
99.0	84.0	1.164153218269348 1E-10	False
99.0	86.0	1.164153218269348 1E-10	False
99.0	46.0	2.910383045673370 4E-11	True
17.0	32.0	0.0	True

Etude 4

Isabel Baltazar 8093074 Lencho Burka 1038530 Nikhil Patil 75669

Is x = f(y) always true for this set of y-values
{0.0, 0.00000000001, 0.02, 10000.0, 500050005.0, π}?
We allowed x and x(n) to have a difference of up to 1E-10.

The formula failed for these y-values: 10000.0, 500050005.

x	y	Difference	Difference < 1E-10?
31.0	500050005.0	31.0	False
84.0	10000.0	1.164153218269348 1E-10	False
97.0	π	1.136868377216160 3E-13	True
83.0	10000.0	0.0	True

If we run the formula using Integer types, then we fail the for almost all n because of rounding errors. If we run the formula using floats or doubles, then we fail fewer times, but we still fail because of rounding errors.

The formula fails at this point: `((x/y) - x(y))`. If y is sufficiently large, then `(x/y)` will be very small but subtracting `x(y)` will make it even smaller.

For example, 31.0/1500050005.0 = 6.1993800000062e-8, a number that is unbelievably small. When you subtract 31.0(500050005.0) = 15501550155 from `(x/y)`, the result is -15501550155, a number which has been truncated so much that the result of `(x/y)` is no longer reflected.

The failures happen more frequently when y > 20. This is because of the `x/y` part of the equation. As y increases, then the the result gets closer and closer to zero. The computer has trouble handling floating point numbers so we lose precision before we get to the end of the formula, resulting in truncation errors.