Ryan Spear 1801851
Kent Loh 5558196

# Numbers

## HARMONIC NUMBERS

Harmonic numbers are the sums of the inverses of integers up to n.
We found that when computing a harmonic number for a given n, the output is always the same up to an n of 60, where computing the harmonic number in order will begin to yield a larger number than computing the harmonic number in reverse. This is true for both Floats and Doubles.

In fact up to an n of 10,000, using a float or a double has no effect on accuracy.
Other than the obvious rounding error, using a float or a double will not have any effect on the output.

```
n: 50.0

Float in order:      4.4992056
Float in reverse:    4.4992056
Double in order:4.499205589294434
Double in reverse:   4.499205589294434
```

When n = 50, we can clearly see accuracy is not affected by variable type nor order of operation.

```
n: 1000.0

Float in order:      7.4854784
Float in reverse:    7.4854717
Double in order:7.485478401184082
Double in reverse:   7.485471725463867
```

When n = 1000, the order of the addition starts to take effect on the accuracy. The difference between float and double at this point is only a rounding difference.

```
n: 10000.0

Float in order:      9.787613
Float in reverse:    9.787604
Double in order:9.787612915039062
Double in reverse:   9.787604331970215
```

Even when n = 10,000, variable type only effects the rounding.
But at this point we can see the order of addition hugely effecting the accuracy of the output.

Ryan Spear 1801851
Kent Loh 5558196

H(11) is equal to H(10) + 1/11.
this means H(11) is equal to 1/10 + 1/11.
We know that may encounter some irrational numbers throughout the incremental steps, so we know that the program will definitely round numbers before adding the next.

This knowledge could explain why the reverse method gives us a different outcome compared to going forward. When working backward, we will be working with the smallest numbers first.

For example, when n = 1000, we start off adding 1/1000 to 1/999 to 1/998, and so on.
These numbers will be much smaller than going forward, and thus creating initial rounding errors from the beginning of our calculation. This creates a domino effect where what would normally be small rounding errors start to stack on top of one another to create differences of 0.000007 at n = 1,000.

How do we know which is more accurate?
Using the popular method of taking the logarithm of n and adding the Euler-Mascheroni constant, we can compare our accuracy.

n: 1000.0

| | |
|---|---|
| Double in order: | 7.485478401184082 |
| Double in reverse: | 7.485471725463867 |
| Euler-Mascheroni: | 7.484970943883669 |

n: 10000.0

| | |
|---|---|
| Double in order: | 9.787612915039062 |
| Double in reverse: | 9.787604331970215 |
| Euler-Mascheroni: | 9.787556036877715 |

We can see here that our algorithms tend to overstate the harmonic number by quite a lot. But in all cases, doing the addition in reverse gives a lower output than forwards. We can say that going in reverse gives a more accurate representation of the true harmonic number.

Ryan Spear 1801851
Kent Loh 5558196

## STANDARD DEVIATION

The standard deviation for a set of numbers can be found two ways, in the following investigation we will refer to the formula:

$$\sigma = \sqrt{\frac{\sum (a_i - \bar{a})^2}{n}}.$$

As formula (1), and formula:

$$\sigma = \sqrt{\frac{\sum a_i^2 - \left(\sum a_i\right)^2 / n}{n}}.$$

As formula (2).

We used a pseudo-random number generator to create a set of 10 numbers from 1000 to 10,000. Running these numbers through differing tests we found many instances of the following output:

```
a: [2234.0, 4938.0, 3102.0, 1307.0, 4185.0, 1209.0, 8976.0, 9232.0, 3456.0]

Double:
(1) Deviation:    2819.3317087477694
(2) Deviation:    2819.3317087477694

Float:
(1) Deviation:    2819.3318
(2) Deviation:    2819.3315
```

In this instance, we see that when using a double there is no difference in output between (1) and (2). It seems as though the pre-computation used in equation (1) does not appear to be a significant player in avoiding rounding issues as formula (2) does the computation as a whole to obtain the same result to 14dp.
Using a float gives us some error here. When we calculate (1) we appear to overstate the standard deviation, and when using (2) we understate the standard deviation, in comparison to the double calculation.

Ryan Spear 1801851
Kent Loh 5558196

Here were are using much larger numbers to stress the bounds of the primate data types of float and double. Running these numbers in two different stress cases, first stressing the float MAX_VALUE of 3.40282347 x 10^38 and then the double MAX_VALUE of 1.7976931348623157 x 10^308, we notice:

**Float's stress case:**

Result of adding a constant 1E18 (found experimentally before the breaking point of float) to a number number between 1-1000, this displays the upper limit of the float. Longer numbers exceed float's MAX_VALUE, which will be demonstrated in the Double stress case.

Double:

(1) Deviation: 9.949874371065952E16

(2) Deviation: 9.9498743710662048E16

Float:

(1) Deviation: 9.9499463E16

(2) Deviation: 9.949875E16

In the above case, with large values bordering the upper limit of the float MAX_VALUE boundary, we find that Double is precise to 12 significant figures (SF). Float, seems to round off much sooner, at 5 SF.

**Double's stress case:**

Result of adding a constant 1E153 (found experimentally before the breaking point of float) to a number number between 1-1000, this displays the upper limit of the float

Double:

(1) Deviation: 9.949874371065977E151

(2) Deviation: 9.949874371066199E151

Float:

(1) Deviation: NaN

(2) Deviation: NaN

Here we see that the deviation, computed with floats, has well exceeded its MAX_VALUE of 3.40282347 x 10^38, resulting in "NaN" (not a number). Double is precise to 14 significant figures in this case. The above case highlights that as numbers get remotely large, doubles too, round which was not found with smaller values where the computations from the two different formulas matched up.

Finally, we added a constant value to each of the numbers in our set (a).
We tried multiple sized constants but found the most interesting output when the constant was large,

pushing the capability of the float and double to close to their maximum. In the following case we used a constant of 100,00.

a: [159909.0, 183858.0, 313735.0, 919781.0, 324854.0, 414419.0, 581715.0, 305767.0, 909662.0, 201478.0]

Double:
(1) Deviation: 268134.78710931935
(2) Deviation: 268134.78710931935

Float:
(1) Deviation: 268134.78
(2) Deviation: 268134.75

Adding fixed amount: 100000.0

Double:
(1) Deviation: 268134.78710931935
(2) Deviation: 268134.78710931964

Float:
(1) Deviation: 268134.78
(2) Deviation: 268133.97

And by increasing the amount of numbers in our set to 1000;

Double:
(1) Deviation: 60499.66876642133
(2) Deviation: 60499.66876641967

Float:
(1) Deviation: 60498.875
(2) Deviation: 60499.67

adding fixed amount: 100000.0
Double:
(1) Deviation: 60499.66876642133
(2) Deviation: 60499.66876641967

Float:
(1) Deviation: 60498.875
(2) Deviation: 60500.18

Ryan Spear 1801851
Kent Loh 5558196

Adding the constant value to each number in the set (a) gives us comparatively large errors than adding no constant for formula (2) only. This is reoccurring when adding large numbers to our set. By looking at the output we can assume that (1) is the better of the two formulas because it is not effected by adding a fixed value. Both using a float and using a double yielded the same result in this test using formula (1).

But to be confident in our assumption that formula (1) is the better of the two we need to try to explain why it doesn't return us errors as (2) does.

After looking over both formulas and looking at our previous experiment result's we think the reason formula (1) performs better is, again, because of rounding errors our float (and sometimes double) creates. Formula (2) squares every number is the set and takes the sum of them. When doing this in the size that we tested, we are guaranteed to get very large numbers, and in turn very large rounding errors.

In formula (1) we only square the difference between the average and $a_i$. With ai being computed separately. Making this crucial difference in each step of the formula, we believe, is the reason the accuracy of method (2) deteriorates once the values in the set become large, and the set size also becomes much larger.

In the situation where we have a large sequence of numbers, and we have a small storage space to utilize. Formula (2) saves storage space because the pre-computation of ai does not need to be stored and computed separately. So if the pre-computation is significantly large, formula (2) could be preferred on the basis of space, time and efficiency.

Ryan Spear 1801851
Kent Loh 5558196

## IDENTITY

The following formula should be equal on both sides of the equals sign:

$$x = \left(\frac{x}{y} - x \times y\right) \times y + x \times y \times y.$$

By looking at this formula and remembering what we have learned from the previous two studies, we have a hypothesis that the last part of the second-hand side, X * Y * Y, will cause us problems when x and y get large. To be specific, when x * y * y > 3.40282347 x 10^38, we expect the float case to break.

Starting low:

x: 300.0, y: 300.0
Double: 300.0
Float: 300.0

where x * y * y = 27,000,000. Not large enough to break the formula according to our hypothesis.
Next we try to break the float formula by using an x and a y that will create a number greater than the 32-bit limit.

x: 1000.0, y: 500.0
Double: 1000.0
Float: 992.0

where x * y * y = 250,000,000. Again, this performs as expected. The float variable cannot contain this value sufficiently and so a rounding error is created, making the float algorithm off by 8.

To test whether this is the reason the float case breaks, we try to use 2 edge case, on just under the limit and one just over.

x: 3.40282347E38, y: 1.0
Double: 3.40282347E38
Float: 3.4028235E38

In this case, x * y * y 3.4028235E38 should be the maximum value our float can handle, the computation rounds this down.

To ensure computation does not round the float down, taking y of 1.1 should give enough of a buffer to exceed this MAX_VALUE of the float and fail.

x: 3.40282347E38, y: 1.1
Double: 3.4028234699999995E38
Float: NaN

Now we have broken the float case, x * y * y =3.40282347E38. This is just greater than what a float can contain. We can safely say that so long as x * y * y is greater than the limit 3.40282347E38 we can expect to break the maximum that the float can hold.
We can also assume that for the MAX_VALUE for a Double being 1.7976931348623157 x 10^308. Any computation that exceeds this maximum value will fail.

So what triggers this behavior is when the MAX_VALUE of the respective data type is exceeded, the computation fails.

There is a very evident issue with rounding for both Doubles and Floats when Y is of a relatively large size and X is of moderate size, not large enough to break the computation. It appears as though, when we break the computation down into LHS and RHS ((x/y) - (x*y) * y) and (x * y * y) respectively). We notice that both the double and the float are rounding to the same SF, roughly 7 SF.

x: 3.7, y: 5.6E15

-1.16032E32 Double lhs

1.16032E32 Double rhs

Double: 0.0

-1.16032E32 float lhs

1.16032E32 float rhs

Float: 0.0

This results in calculated X value, performed by the operation (((x/y) - (x*y)) * y) + (x * y * y), to be 0.0. The expected x value, however, should be 3.7. This rounding issue unusually happens in both Double and Float computation, we will explore the point in which this computation is maintained to give a correct X value for doubles, because we previously examined doubles to round much later on.

x: 3.7, y: 5.6E7

-1.1603199999999996E16 Double lhs

1.16032E16 Double rhs

Double: 4.0

-1.16031999E16 float lhs

1.16031999E16 float rhs

Float: 0.0

Given Y with a magnitude of 5.6E7, double gives a value of 4.0, which is not perfect (3.7) but is closer than the result of 0.0 for larger values of Y. This demonstrates that given the number of 5.6E7, the LHS computation clearly gets rounded off much later displaying 18 SF. The key idea is that Float and Double seems to employ different rounding or storage techniques depending on the computation, more likely based on the limitation of time where speed takes precedence over some form of accuracy and precision.

The below example explores a much bigger magnitude of Y to be in the scope of Double's upper bounds. Rounding still seems to behave the same on the LHS and RHS, however the rounding of the Float computations goes to infinity as it has well exceeded the maximum it can store.

```
x: 3.7, y: 5.6E153

-1.16032E308 Double lhs

1.16032E308 Double rhs

Double: 0.0

-Infinity float lhs

Infinity float rhs

Float: NaN
```

A magnitude higher, Y of 5.6E154, results in the Double computation (both LHS and RHS) going to infinity and the resulting number cannot be computed.

```
x: 3.7, y: 5.7E154

-Infinity Double lhs

Infinity Double rhs

Double: NaN

-Infinity float lhs

Infinity float rhs

Float: NaN
```

All in all, these rounding techniques of Doubles and Floats do not seem to deal with computations of large values of Y and moderate values of X. Rounding is often restricted to 8 SF, leading to a result of 0.0, truncating the actual computations significantly to appear as though, on the LHS, multiplying a negative big number by another big number seems to be negligible due to truncating and rounding at 8 SF. There have been clear differences in our expected X value and computed X value due to these seemingly shortcuts in computation that prioritize speed over accuracy.