**Etude 6: Numbers**

**Harmonic Series:**

*Note: BD = BigDecimal, D = Double, F = Single (Float). The underlining in the table identifies the significant digits when compare the two implementation results against the BigDecimal reference value.*

|  |  |  |
| --- | --- | --- |
| ***n*** | **Non-reversed** | **Reversed** |
| **1** | 1.0 (F)  1.0 (D)  1.0 (BD) | 1.0 (F)  1.0 (D)  1.0 (BD) |
| **10** | 2.9289684 (F)  2.9289682539682538 (D)  2.9289682539682539430536678537464467808604240417480468750 (BD) | 2.9289684 (F)  2.9289682539682538 (D)  2.9289682539682539430536678537464467808604240417480468750 (BD) |
| **100** | 5.187378 (F)  5.187377517639621 (D)  5.18737751763962021307741157016835131798870861530303955078125 (BD) | 5.187377 (F)  5.1873775176396215 (D)  5.18737751763962021307741157016835131798870861530303955078125 (BD) |
| **1000** | 7.4854784 (F)  7.485470860550343 (D)  7.48547086055034486119612313625992783272522501647472381591796875 (BD) | 7.4854717 (F)  7.485470860550341 (D)  7.48547086055034486119612313625992783272522501647472381591796875 (BD) |
| **10,000** | 9.787613 (F)  9.787606036044348 (D)  9.787606036044382210194571627970283600461698370054364204406738281250 (BD) | 9.787604 (F)  9.787606036044386 (D)  9.787606036044382210194571627970283600461698370054364204406738281250 (BD) |
| **100,000** | 12.090851 (F)  12.090146129863335 (D)  12.090146129863427893276126273008896916394405707251280546188354492187500 (BD) | 12.090153 (F)  12.090146129863408 (D)  12.090146129863427893276126273008896916394405707251280546188354492187500 (BD) |
| **1,000,000** | 14.357358 (F)  14.392726722864989 (D)  14.392726722865723577218399385161534675958705520315561443567276000976562500 (BD) | 14.392652 (F)  14.392726722865772 (D)  14.392726722865723577218399385161534675958705520315561443567276000976562500 (BD) |

Table 1.

**Do both answers for single and double precision implementation agree with each other?**

Yes, results for both implementations do agree with each other. However, they do have a slight difference at their least significant digits. This difference tends to increase as the size of *n* gets larger. Also there is an obvious difference in the level of precision for both data types. Single point precision has fewer significant decimal digits than doubles.

**Does the answers to reverse implementation agree with the normal implementation?**

Based on the results in table 1 the reversed and normal implementation results are similar. But again we start seeing some differences at the least significant digits as we start getting higher values of *n*.

**If not the same, which implementation is more accurate and how can we test this?**

Results from both implementations are similar but not exactly the same. Table 1 shows the reversed implementation is more accurate the than the normal way. We can test this using the BigDecimal class in Java which allows us to compute numbers with arbitrary precision. We computed the results for both implementations using the BigDecimal and then compared them to the results we obtained from the double implementations. Both results are similar up to n = 1000 - by similar we mean the least significant digit only differs so there is not much difference overall. But after n = 1000, more significant digits are different different compared to the reference answer (labelled BD). Although the difference isn’t much, if we were to pick the better out of both implementations the reversed implementation gets closer results to the reference value.

**Standard Deviation:**

*Note: Will name normal standard deviation as the two-pass standard deviation and the non-general standard deviation as the one-pass standard deviation.*

DataSets:

1. 3, 6, 12, 20, 33, 40, 60, 100, 30, 53
2. 0, 1, 12, 20, 33, 40E30, 60E9, 100E70, 30E50, 53E60
3. 13E-50, 12E-60, 17E-20, 33E-15, 40E30, 60E9, 100E70, 30E50, 53E60
4. 13, 12E60, 17E20, 33E15, 40E30, 60E9, 100E70, 30E50, 53E60
5. 1.3, 12E-60, 17E-20, 33E-15, 40E-30, 60E-9, 100E-70, 30E-50, 53E-60
6. Random 1,000,000 data points with seed set to 12000.

*Key: (O) - offset of 1E100.*

|  |  |  |
| --- | --- | --- |
| **Dataset** | **Two-Pass Results** | **One-Pass Results** |
| **1** | 28.003749748917553  1.942668892225729E84 (O) | 28.003749748917556  NaN (O) |
| **2** | 2.9999999999823335E71  1.942668892225729E84 (O) | 2.999999999982333E71  NaN (O) |
| **3** | 3.1426968052527244E71  0.0 | 3.142696805252724E71  NaN (O) |
| **4** | 3.14269680524801E71  0.0 | 3.14269680524801E71  NaN (O) |
| **5** | 0.40855058232853736  0.0 | 0.40855058232853736  NaN (O) |
| **6** | 0.2887070741478686  1.0800267706328941E89 (O) | 0.28870707414786256  NaN (O) |

Table 2.

**Does the two standard deviation implementation agree with each other?**

The two standard deviation implements may differ by the least significant digit, but overall the two standard deviations results very similar. Table 2 shows that both standard deviation methods practically have the same answers - values without ‘O’ key label.

**Does the answers to your implementation stay the same when adding a fixed value to all sequence of numbers and does the level of precision matter?**

No adding a fixed value to the sequence of numbers doesn’t necessarily keep the answer the same. These results are shown in table 2 for the values with the ‘O’ key label. Up to an offset of approximately 1,000,000 the results are the same, but any value beyond that limit results in invalid values like the ones shown in table 2. NaN occurs because powering very large numbers will exceed the double binary representation limit. We get zeros because of addition and subtraction issue related to the precision limit (significant decimal precision). This shows that for data points in a dataset, the level of precision matters. We can’t handle extremely small or large numbers because eventually we could be adding or subtracting a very small number from a very large number or vice versa - this is explained in the *identity* section further below.

**Which standard deviation is generally preferred?**

The two-pass formula is preferred over the one-pass standard deviation formula if memory, and velocity of data is not an issue. This is because the one-pass standard deviation formula could take the square root of a negative number (Knuth, 1969). There is a better standard deviation formula from Welford’s variance computation using recurrence equations but we will stick the assignment scope. Also depending on context, table 2 shows the two-pass implementation tends to keep an extra level of precision - one extra significant digit.

**Are there any situations where the non-general standard deviation is favoured?**

The one-pass standard deviation formula is favoured when data is constantly changing, and when we need to recompute the standard deviation for new data. An extreme example like big data from the Large Hadron Collider generates 1 petabyte of data per second - high velocity (CERN, 2017). If we were to compute the standard deviation of a variable, the two-pass standard deviation formula will be slow. This is slow because for big data we have to store it in memory and read it twice. We have to read the data twice because in the first pass we compute the mean, and then on the second pass we compute the squared residual differences - this is time consuming for processing big data. Using the one-pass standard deviation, we can speed this process because we don’t need to precompute the mean (Welford, 1962).

**Identity:**

**Can you find any where it apparently fails?**

In implementation, fractions that are recurring will fail with the identity formula. But the answer practically is the same because it doesn’t differ by much. The formula completely fails when the x-value is extremely small, and the y-value is extremely large. If this is the case, and when we compute both x/y and x\*y and subtract both answers, we get zero which causes the formula to fail.

**What triggers this behaviour?**

The formula fails in implementation because both single point and double point precision have a finite space to represent precision (significant decimal places). Using the IEEE 754 standard single point precision can represent 7 and double point precision can represent 15 significant decimal places. For addition and subtraction floating point arithmetics, both numbers must be scaled to the same exponent. Because of this numbers outside the precision range will be truncated - so we lose information.

For example say we want to subtract 0.01f to 1000000.1f:

In single point precision, the binary representation of 1000000.1f is: 11110100001001000000.001.

In single point precision, the binary representation of 0.01f: 0.00000010100011110101110000101.

To add these two numbers we have to subtract at the same precision.

So it will look something like this:

11110100001001000000.00100000000000000000000000000

- 0.00000010100011110101110000101

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

But because for a single point precision we only have 32-bits to represent our number, our subtraction will look more like this:

11110100001001000000.001

- 0.000

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Essentially we are subtracting 0f from 1000000.1f because we have lost the least significant bits. Java also agrees and prints out 1000000.1f.

**Reference:**

CERN Accelerating Science. (2017). *Cern Data Centre passes the 200-petabyte milestone.* [Online] Available from: https://home.cern/about/updates/2017/07/cern-data-centre-passes-200-petabyte-milestone [Accessed 4th February].

John D. Cook Consulting. (2008). *Comparing three methods of computing standard deviation.* [Online] Available from: https://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation/

Welford, B. P. (1962). *Note on a Method for Calculating Correct Sums of Squares and Products.* Technometrics, 4 (3), pp. 419-420. [Online] Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.7503&rep=rep1&type=pdf [Accessed 4th February].

Knuth, D. E. (1969). *The Art of Computer Programming: Vol 2 - Seminumerical Algorithms.* [Online] Stanford, Addison-Welsey. Avaliable from: https://doc.lagout.org/science/0\_Computer%20Science/2\_Algorithms/The%20Art%20of%20Computer%20Programming%20%28vol.%202\_%20Seminumerical%20Algorithms%29%20%283rd%20ed.%29%20%5BKnuth%201997-11-14%5D.pdf

Wolfram MathWorld. (2018). *Roundoff Error.* [Online] Available from: http://mathworld.wolfram.com/RoundoffError.html [Accessed 4th February].

Stack OverFlow. (2011). *Adding 32 bit floating point numbers.* [Online] Available from: https://stackoverflow.com/questions/7884343/adding-32-bit-floating-point-numbers [Accessed 4th February].