1 Introduction

1.1 Motivation

Stochastic rounding maps a real number to one of its floating-point bounds, with probabilities based on the relative distance of those bounds.

This report analyses three different applications of stochastic rounding, implemented based on the definition presented in the second week of this unit.

$$\mathrm{SR}(x) = \begin{cases} \mathrm{RA}(x), & \text{if } P < \frac{x-\mathrm{RZ}(x)}{\mathrm{RA}(x)-\mathrm{RZ}(x)} =: p \\ \mathrm{RZ}(x), & \text{if } P \geq p \end{cases} \quad (1)$$

where $x \in \mathbb{R}$, RA stands for round-away-from-zero, RZ for round-toward-zero, and $P \in [0, 1)$ is a randomly generated number from a uniform distribution.

1.2 Machine specifications

All of the following experiments are done on a Lenovo Legion 5 with the following specifications.

Machine Specifications	
OS	Ubuntu 22.04.3 LTS x86_64
CPU	AMD Ryzen 7 5800H with Radeon G
GCC	(Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0
FMA	enabled
80-bit	enabled

Table 1: Specifications of experiment machine

The C file was compiled with GCC by running `gcc -O3 -march=native -mfma assignment1.c -lm`

While FMA is enabled, the generated binary doesn't have any FMA instructions. The execution time on this machine is around `25s`.

2 Part I: Implementation and testing of binary32 stochastic rounding

2.1 Implementation

To implement stochastic rounding as defined in 1, we need to calculate RA and RZ, then solve for the probability p, generate the random number P, and use it to choose what to return.

2.1.1 Finding RA and RZ

These values lie on the FP bounds of a real number. Their order depends only on the sign of x. To compute the FP bounds we use `<math.h>`, which includes a suite of functions[6] for dealing with neighbouring FP numbers. `nextafterf(f,±INFINITY)` will return the next or previous floating-point number depending on the direction given. Using `nextafterf`, **algorithm 1** can be constructed. It works by first computing the nearest floating-point number to x. This number has to be a bound, depending on which it is (upper or lower), the other is computed. Then according to the sign of x, RZ and RA are computed.

The number of operations done to compute the bounds can be reduced since, in algorithm 1, one of the calls to `nextafterf` will inevitably not be used as it isn't a bound of x. In **algorithm 2** we take advantage of the fact that casting to float is guaranteed to give one of the bounds, we can use another function, `nexttowardf`, in the direction of x to get the opposite bound. With the bounds computed, depending on which is larger and the sign of x, RA and RZ are set.

```c
void RA_RZ(double x, float *ra, float *rz) {
  float f = (float) x;
  float hi = nextafterf(f, INFINITY);
  float lo = nextafterf(f, -INFINITY);

  if(f > x)
    hi = f;
  else if(f < x)
    lo = f;
  else {
    *ra = f;
    *rz = f;
    return;
  }

  if(x > 0.0) {
    *ra = hi;
    *rz = lo;
  } else {
    *ra = lo;
    *rz = hi;
  }
}
```

Algorithm 1: RA & RZ method 1.

```c
void RA_RZ(double x, float *ra, float *rz) {
  float f1 = (float) x;
  float f2 = nexttowardf(f1, x);

  if (f1 > f2 == x > 0.0 ){
    *ra = f1;
    *rz = f2;
  } else {
    *ra = f2;
    *rz = f1;
  }
}
```

Algorithm 2: RA & RZ method 2.

2.1.2 Stochastic rounding

With RA and RZ computed, implementing stochastic rounding is simply a matter of generating the random number $P \in [0, 1)$, computing the probability p, and selecting one of the 2 bounds according to equation 1. This is shown in **algorithm 3**. While not present in the equation, the case where x has a direct representation in floating-point needs to be handled. If $x = \text{RN}(x)$, both methods 1 & 2 will give $\text{RZ}(x) = \text{RA}(x) = x$. To avoid dividing by 0, algorithm 3 simply returns $\text{RA}(x)$.

```
1  float SR_alternative(double x) {
2    float ra, rz;
3    RA_RZ(x, &ra, &rz);
4
5    if(ra == rz)
6      return ra;
7
8    double p = (x - rz) / (ra - rz);
9    double P = (double) rand() / RAND_MAX;
10   return (P < p) ? ra : rz;
11 }
```

Algorithm 3: Stochastic rounding.

2.1.3 Note on random numbers

A crucial aspect of the equation 1 is that the random number P is sampled from an uniform distribution. This is mentioned by Michael Hopkins et al.[2] "*for SR to be competitive in terms of computation time a very efficient source of high-quality pseudo-random numbers is critically important*". In our implementation P is computed by `(double)rand()/RAND_MAX`. While `rand()` does generate pseudo-random[3] numbers from an uniform distribution, `(double)rand()/RAND_MAX` will not cover all double (`IEEE 754`) numbers in the range $[0, 1)$. Moreover, even if we would generate all possible random doubles in in the range $[0, 1)$, they would still not even begin to cover the infinity of reals in the same range.

Therfore, the fundamental aspect is that the probability distribution of the attainable values is uniform, with these values being evenly distributed across the range, and not concentrating around any particular value.

2.2 Validation

In order to validate the alternative implementation of stochastic rounding, we repeatedly round a value that is representable in binary64 more accurately than in binary32, and we gather 2 forms of statistics:

1. Compare the **probability** of the algorithm of rounding up or down, with the probability of the benchmark implementation and the expected analytical probability.
2. Compare the **average** values as the number of rounding increase of method 3 and the benchmark algorithm against the value of the binary64 number.

The default parameters for validation were used, the number π (in binary64) was rounded 5 million times.

2.2.1 Rounding probability

The probability of rounding up was used, at each iteration the result of the rounding operations were compared to the high or low bounds of the sample number and recorder in counters. At the end of the iteration the probability of rounding up was calculated using equation 2 and the expected probability was calculated using equation 3.

$$p = \frac{\#\text{high}}{\#\text{high} + \#\text{low}} \tag{2}$$

$$\hat{p} = \frac{\text{high}_x - x}{\text{high}_x - \text{low}_x} \tag{3}$$

The following are the results generated. It is worth noting that while the code deals with randomness these results do not vary significantly and are representative for the performance of the method.

expected	benchmark	ours
0.633322	0.633830	0.633333

Table 2: Probability of selecting the upper bound

2.2.2 Average values

The absolute errors of the average values were recorded at every 1000th iteration. The absolute error at iteration `i` is given by this code, where `K` is the total number of iterations and `avg`* is the sum of the first `i` elements divided by `K`

$$\texttt{err}_i = |\ \texttt{sample - avg}^* \texttt{ * K / i}| \tag{4}$$

These values are then saved[A] to a file and plotted in a Jupyter Notebook[B].

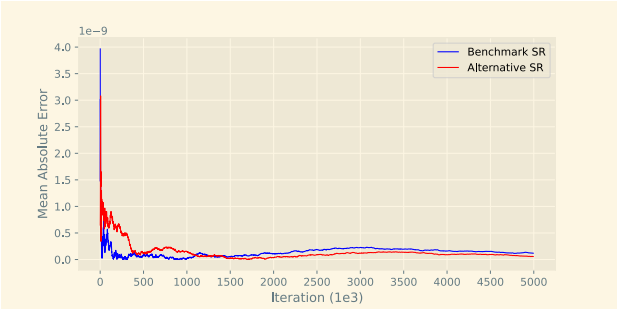

Figure 4: Absolute error of average of SR overtime

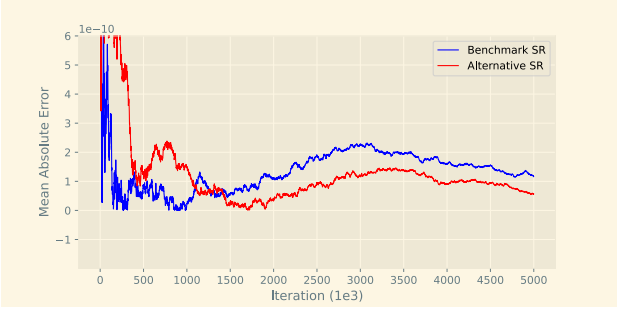

Figure 5: Plot 4 zoomed in

2.3 Analysis

Both validation tasks prove the out alternative implementation of stochastic rounding is **performing as expected**. The probability of choosing the upper bound is accurate within 3-4 decimal places, as is the benchmark implementation. The curve of the absolute error of average values over iterations closely matches the benchmark, even very slightly outperforming it after 1.5 million iterations, though this is not a statistically significant improvement as on different seeds the graph can look inverted.

It is worth considering the **speed performance** of this method compared to the benchmark, since it introduces *more* operations and branching. While the two algorithms seem practically equivalent, the benchmark implementation has fewer operations and no branching, and so it is more efficient. Trying to model the logic in equation 1 will inevitably produce more operations. Since the outcome is almost identical, in a performance critical application the benchmark implementation would be better suited.

3 Part II: Stagnation and the Harmonic series

We compute a truncated version of the harmonic series using the four following methods and analyse the results:

1. Recursive summation with binary32 arithmetic and round-to-nearest mode (default implementation)
2. Recursive summation with binary32 arithmetic and **stochastic rounding**
3. Compensated summation with binary32 arithmetic and round-to-nearest mode (`fastTwoSum`)
4. Recursive summation with binary64 arithmetic and round-to-nearest mode (reference implementation)

Stagnation occurs when increasingly small FP numbers are added to increasingly large ones until $\text{RN}(a+b) = a$, where $a \gg b$. This section tests methods that try to overcome the issue of stagnation. Stochastic mitigates stagnation by introducing the chance to round up ($\text{RA}(x)$) even when $\text{RN}(a+b) = a$. In contrast, compensated summation methods increase accuracy by keep track of a compounded error term which is added back into the sum.

3.1 Implementation

Methods 1 and 4 simply add the cast the next iteration in the harmonic series term to FP and add it into the sum. Stochastic rounding is used in method 2 by casting the sum to binary64 when adding the current term in the series, then using SR to cast it back to binary32. Compensated summation, used in method 3 uses the `fastTwoSum`[C] function showed in Week 2 to update the sum and the error term.

```
1  fharmonic += (float)1/i;
2  fharmonic_sr = SR_alternative((double)fharmonic_sr + (double)1/i);
3  fastTwoSum(fharmonic_comp, (float)1/i + t, &fharmonic_comp, &t);
4  dharmonic += (double)1/i;
```

Algorithm 6: Truncated Harmonic Series

3.2 Validation

In order to verify the correctness of the methods' implementations, the computed sums are compared to the reference, binary64 round-to-nearest, truncated harmonic sum. Using the first 500 million terms of the harmonic series, the results are presented in table 3

RN	SR	fast2sum	**binary64**
15.4036827	**20.607**4790	**20.6073341**	**20.6073343**

Table 3: Truncated harmonic sum results

and the corresponding absolute errors in table 4.

RN	SR	fast2sum
5.2036516	0.0001447	0.0000001

Table 4: Truncated harmonic sum errors

These results clearly demonstrate that both stochastic rounding and compensates summation give results that closely match the higher precision reference, validating their implementations. It is also apparent that compensated summation gives the better approximation for this particular task. Furthermore it is apparent that the recursive summation with binary32 arithmetic and round-to-nearest produces inaccurate results. The code also measures that it **stagnates at i 2,097,152**, in the first 0.004 % of iterations.

3.2.1 Absolute Error

The absolute errors of the sums were recorded at every 1 million iterations. The absolute error at iteration i for method m is given by this relation

$$\texttt{err}_i^m = |\ \texttt{dharmonic}_i\ \texttt{-}\ \texttt{fharmonic}_i^m| \qquad (5)$$

These values are then saved[A] to a file and plotted in a Jupyter Notebook[B] using a logarithmic scale.

Figure7 confirms the trend taking shape in validation where at every time step, compensated summation is the best approach followed by stochastic rounding, and finally round-to-nearest. The point were the RN sum stagnates can also be clearly seen on the graph when the curve suddenly becomes smooth since the only change in the absolute error comes from the reference sum increasing.

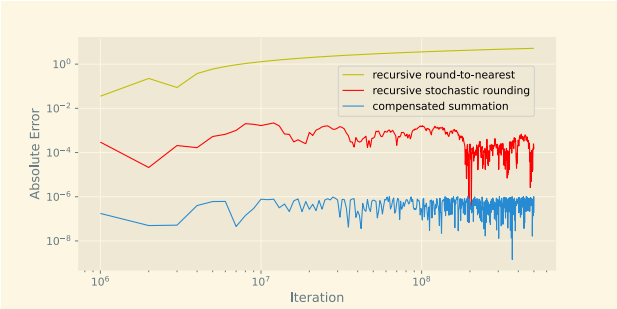

Figure 7: Absolute error of harmonic series overtime

3.3 Analysis

It seems clear that the best approach for computing the truncated harmonic series is compensated summation. This is to be expected, since it *directly* tackles the problem of error correction by reducing compounded error. In contrast, stochastic rounding mitigates compounded error in a probabilistic manner, which is less effective for this application.

Additionally, once again our implementation of SR incurs some performance overhead compared to the `fastTwoSum` version. The only benefit of SR in this application is that it doesn't require maintaining a compounded error term.

4 Part III: The Riemann zeta function and convergence

In this section we are going to analyze the performance of SR on convergent series. We will approximate the Riemann zeta function at $x = 2$. This was shown to converge to $\frac{\pi}{6}^2$ by Euler in 1741[1].

$$\zeta(x) = \sum_{i=1}^{\infty} \frac{1}{i^x} = \frac{1}{1^x} + \frac{1}{2^x} + \frac{1}{3^x} + \cdots \qquad , x \in [1, \infty) \tag{6}$$

$$\zeta(2) = \frac{\pi^2}{6} \tag{7}$$

We will use the same summation methods as in Part II, and compare their performance in relation to the binary64 reference sum.

4.1 Implementation

The implementation is nearly identical to Part II, only `i` becomes `i*i`. We should note that the type of `i` should now be `long int` since we need to be able to store its square until iteration 500 million.

4.2 Results

We store[A] the absolute error of each method every 100,000 iterations and plot[B] the results in a Jupyter Notebook. The sums calculated after 500 million iterations are shown in table 5

RN	SR	fast2sum	**binary64**
1.644725322	1.**644937**515	**1.644934058**	**1.644934057**

Table 5: Truncated Riemann zeta results

and the corresponding absolute errors in table 6.

RN	SR	fast2sum
0.0002087351	0.0000034574	0.0000000003

Table 6: Truncated Riemann zeta errors

These results are consistent with the outcomes in Part II, compensated summation produces the best results, followed by stochastic rounding. This is also consistent with the absolute error tracked over iterations, shown in figure 8.

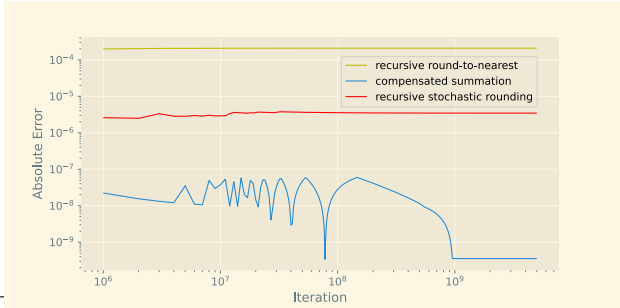

Figure 8: Absolute error of Riemann zeta overtime

4.3 Analysis

Once again, the most accurate method proves to be compounded summation. This means that across the board, whether a truncated infinite series diverges or converges, `fastTwoSum` is the more precise method. Stochastic rounding seems to sit right in the middle of RN and compensated summation. These experiments show that stochastic rounding is a balance of memory, speed and accuracy.

5 Conclusion

In this report we have analyzed 3 applications of stochastic rounding: repeated rounding, summing truncated divergent series and summing truncated convergent series. Overall, SR is more accurate than RN in all of these scenarios, but less accurate then compensated summation. SR is best suited for applications where memory efficiency is critical, such as machine learning[5], or when modeling sensitive systems which suffer from the rounding bias of FP numbers, such as neural ordinary differential equations[2], or even for systems with inherently probabilistic mechanics such as quantum computing[4].

References

[1] Raymond Ayoub. "Euler and the Zeta Function". In: *The American Mathematical Monthly* 81.10 (1974), pp. 1067–1086. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2319041.

[2] Michael Hopkins et al. "Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2166 (2020), p. 20190052. DOI: 10.1098/rsta.2019.0052. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2019.0052.

[3] *ISO/IEC 9899:2018 draft*. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

[4] Rajiv Krishnakumar and William Zeng. *Quantum Rounding*. 2021. arXiv: 2108.05949 [quant-ph].

[5] Lorenz K Muller and Giacomo Indiveri. "Rounding methods for neural networks with low resolution synaptic weights". In: *arXiv preprint arXiv:1504.05767* (2015).

[6] CPP reference. *nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl*. URL: https://en.cppreference.com/w/c/numeric/math/nextafter.

Contents

Appendices

A Saving Data

The generation and saving of statistics to a file is determined by the macro constants `SAVE_PART1` and `SAVE_PART2`, they should be defined to the desired name of the output file and the program will save the data in those files. This choice was made to avoid unnecessarily allocating stack memory and perform other unnecessary checks when we are computing statistics.

In this projects archive we have included a folder `data/` which contains all the raw data generated and used in this report.

B Plotting

The plots where created in the provided Jupyter Notebook `Plotting.ipynb`. The following is the script for plotting the Part I data. The rest can be found in the notebook and are very similar.

```
k = len(data)
xs = np.arange(0, k * 1000 + 1, 1000)
fig, ax = plt.subplots()

plt.plot(xs[1:], data_def, "b", linewidth=1, label="Benchmark SR")
plt.plot(xs[1:], data_alt, "r", linewidth=1, label="Alternative SR")

plt.legend()
plt.gcf().set_size_inches(8, 4)
plt.gcf().set_dpi(100)
plt.xticks(xs[::500])
ax.xaxis.set_major_formatter(FuncFormatter(lambda x, pos: f"{x // 1000}"))
ax.set_ylim(top=6e-10) # remove this for non-zommed in version
plt.xlabel("Iteration (1e3)")
plt.ylabel("Mean Absolute Error")

# plt.savefig('part1_zoom.svg')
```

C `fastTwoSum`

This function is used for compensated summation in Part II, it is implemented directly from the Week 2 demonstration as follows.

```
void fastTwoSum (float a, float b, float *s, float *t) {
  float temp;

  *s = a + b;
  temp = *s - a;
  *t = b - temp;
}
```