

# COMP36212 Mathematical Systems and Computation 2020/21

---

Week 3: Extending the Precision

Mantas Mikaitis, Department of Mathematics

Email: [mantas.mikaitis@manchester.ac.uk](mailto:mantas.mikaitis@manchester.ac.uk)

# Goals

- In the previous week, floating-point arithmetic of precision  $p$  was presented.
- We call precision  $p$  a *working precision* of a computer (most commonly this is the precision that hardware computes in).
- The main goal this week is to learn of the ways to extend the accuracy of results beyond the working precision.
- There are two strategies of interest:
  1. Use hardware precision  $p$  to obtain more accurate results as unevaluated sum of two or more values in precision  $p$ .
  2. Use software to compute in higher precision than  $p$ .

# Literature

The following material, and the references therein, can be used for further reading.

- *Handbook of Floating-Point Arithmetic* by J.-M. Muller and others, 2<sup>nd</sup> edition, 2018 ([HFP18](#)).
  - Sections 4.3 and 4.4 on 2Sum, Fast2Sum and 2MultFMA algorithms.
  - Section 5.3 on accurate summation algorithms.
  - Most of Chapter 14.
- Various useful details on summation algorithms are in Chapter 4 of *Accuracy and Stability of Numerical Algorithms* by N. J. Higham, 2<sup>nd</sup> edition, 2002 ([ASNA02](#)).

# Contents

- Mixed-precision arithmetic and algorithms.
- 2Sum, Fast2Sum, and 2MultFMA algorithms.
- Summation algorithms: recursive, compensated, and cascaded summation.
- Arbitrary-precision arithmetic libraries.
- Stochastic rounding: motivation, usage, and advantages in summation.

# Some remarks

- As in the previous weeks, we refer to floating point with the abbreviation “FP”.
- We refer to FP operations with  $\circ(x \text{ op } b)$  with  $\circ \in \{\text{RN}, \text{RZ}, \text{RU}, \text{RD}\}$  and  $\text{op} \in \{+, -, \times, /, \sqrt{\cdot}\}$ .
- We refer to exact operations with  $x \text{ op } b$ .
- We use the *italic* font to emphasize the important terms.
- We use the `typewriter` font for algorithms and code.
- For the multiplication we use  $*$  in the algorithms and  $\times$  in the equations.

# Mixed-precision

- Mixed-precision arithmetic can be understood in two ways:
  - ❑ Mixed-precision operations—an adder, a multiplier, or other, that produces an answer by using multiple precisions (internally, or input/output).
  - ❑ Mixed-precision algorithms—algorithms that utilize operations of different (finite) precisions at different steps.
- Note the different terminology used in literature: mixed precision, arbitrary precision, variable precision, multiple precision, infinite precision—the first concept differs from the other four.

# Mixed-precision

- An arithmetic operator with the same precision in inputs/outputs is called a *homogeneous* operator.
- An arithmetic operator with different precisions in inputs/outputs is called a *nonhomogeneous* operator.
- Most processors today implement only homogenous arithmetic operations.
- IEEE 754 (1985) did not include a requirement for nonhomogeneous variants.
- Later iterations of the standard include them, but hardware has not yet caught up.

# Mixed-precision: FMA

- One example of mixed-precision is the FMA instruction, that appears in most modern CPUs (covered briefly last week).
- Given three FP precision- $p$  numbers  $a$ ,  $b$ , and  $c$ , the FMA instruction computes  $\text{RN}(a \times b + c)$ .
- Since by definition there can only be one rounding error in the whole computation, the result of  $a \times b$  is not rounded before the addition.
- It is held exactly in a wider internal format of  $2p$  bits; in this format the addition of  $c$  is performed followed by rounding back to  $p$ .

# Mixed-precision in algorithms

Consider adding  $2^{-24}$  a hundred million times to a variable initialized to 1 in binary32 arithmetic (see example `mixed_precision_example0.c`).

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     float sum = 1;
6     float addend = pow(2, -24);
7
8     for (int i = 1; i < 1000000000; i++)
9         sum = sum + addend;
10
11    printf("%.30f \n", sum);
12 }
```

- In this example we do not have any mixed precision.
- All the variables are `float`.
- This basic computation should add the small value a hundred million times to 1 and return  $\sim 6.96$ .
- However, it returns 1.
- The problem is that `addend` is too small to be added to 1 in binary32 arithmetic—the sum is rounded to 1 on each iteration.

# Mixed-precision in algorithms

We can fix this problem by introducing higher precision in the addition step (see `mixed_precision_example1.c`).

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     float sum = 1;
6     float addend = pow(2, -24);
7
8     double temp_sum = sum;
9
10    for (int i = 1; i < 100000000; i++)
11        temp_sum = temp_sum + addend;
12
13    sum = (float)temp_sum;
14
15    printf("%.30f \n", sum);
16 }
```

- On line 8, we convert sum to binary64 precision temporarily.
- On line 11, we perform binary64 addition.
- On line 13 we convert the temporary sum back to binary32 precision.
- The answer produced by this program now is **6.96046 ...**
- Both programs produce a binary32 answer, but one produces a completely wrong answer.

# Optional exercise

- Modify `mixed_precision_example0.c` to produce the right answer.
- Do not use mixed-precision—only use the `binary32` (`float`) variables.
- The exact answer is 6.9604644775390625.
- Once done, compare with the solution in `mixed_precision_example2.c`.

# Mixed-precision in algorithms

- But note that the technique showed above can impact performance if binary64 arithmetic is more expensive than binary32 (depends on the processor that is used).
- Other direction that is sometimes taken is to reduce the precision in different parts of a program or algorithm.
- If a computer has fast low-precision arithmetic, find the parts in your algorithms that can leverage it without causing major errors.
- Therefore, mixed-precision algorithms are used either to improve accuracy or performance.
- This kind of programming requires very good knowledge of the underlying hardware.

# 2Sum and Fast2Sum algorithms

- Given two FP precision- $p$  numbers,  $a$  and  $b$ , we can obtain  $s$  and  $t$  such that  $s = \text{RN}(a + b)$  and  $s + t = a + b$ .
- Here  $s$  and  $t$  are precision- $p$  FP numbers.
- These methods are called *error-free transformations*.
- Note that  $\text{RN}(x)$  refers to *round-to-nearest ties-to-even* rounding mode, which is a default mode on most processors.
- Other rounding modes cannot assure the error-free property.
- We can think of  $s$  as the answer we get from the FP addition, and  $t$  as the rounding error in the addition.
- When a tuple  $(s, t)$  represents one quantity, we say that it is represented as an *unevaluated sum* of two floating-point values.
- Note that computing the sum of  $s$  and  $t$  does not achieve anything, since  $\text{RN}(s + t) = s$ , so it is only useful to have them separately.

# 2Sum and Fast2Sum algorithms

Algorithm 3.1: Fast2Sum

```
inputs a, b, with |a| ≥ |b|
s = RN(a + b)
b' = RN(s - a)
t = RN(b - b')
return (s, t)
```

- Here is one algorithm that performs an error-free transform.
- Note that arguments have to be sorted by magnitude.
- The rounding mode must be RN, otherwise  $t$  would not be an *exact error*.
- However, even if the two requirements are violated, the sum of  $s$  and  $t$  might still be good a approximation to the sum of  $a$  and  $b$ .
- Fast2Sum is useful in *compensated summation* algorithms that take inputs sorted in any way (see later slides).

# Fast2Sum informal explanation

Algorithm 3.1: Fast2Sum

```
inputs a, b, with |a| ≥ |b|
s = RN(a + b)
b' = RN(s - a)
t = RN(b - b')
return (s, t)
```

- First step is to simply add the arguments.
- In that addition, we lose\* some part of  $b$  to rounding.
- The second step obtains the actual, rounded  $b, b'$ , that was used in the addition.
- The final step computes the size of a piece of  $b$  that was lost in the first step.

\* Or add something to  $b$ , depending on the rounding direction taken by RN.

# 2Sum and Fast2Sum algorithms

Algorithm 3.2: 2Sum

```
inputs a, b
s = RN(a + b)
a' = RN(s - b)
b' = RN(s - a')
ea = RN(a - a')
eb = RN(b - b')
t = RN(ea + eb)
return (s, t)
```

- Here is a more robust version of the previous.
- Advantage over Fast2Sum is that no sorting of arguments is required.
- Similar principle, but now either  $a$  or  $b$  can have smaller magnitude.
- The algorithm does not assume which.
- In steps 2 and 3, either  $a' = a$  or  $b' = b$ .
- Therefore only one of  $e_a$  or  $e_b$  can be nonzero.
- The algorithm has  $2\times$  more steps, but the *depth* is 5—steps 4 and 5 are parallel.

# Optional exercise

- Check `2sum.c` in the example code where two `floats` are added with the two presented algorithms.
- Run it and observe the outputs.
- Notice what results `Fast2Sum` computes when arguments are swapped.
- Check that  $s \neq a + b$ .
- Confirm with more precision (for example, on paper) that we have performed an error-free addition by checking that

$$s + t = a + b.$$

# 2MultFMA algorithm

Algorithm 3.3: 2MultFMA

```
inputs a, b  
s = RN(a * b)  
t = RN(a * b - s)  
return (s, t)
```

- Now we look at the multiplication.
- This requires the FMA instruction (see week 2) for performing the second step.
- First step performs a basic multiplication operation.
- The result from step 1 is a rounded multiplication result.

- In step 2, the FMA is used to compute the multiplication again (but recall that without rounding) and subtract the rounded result.
- Thus,  $t$  is the error induced in step 1, and  $a \times b = s + t$ .
- But there are some exceptions:  $t \neq a \times b - s$  for very small  $a$  and  $b$  (underflow).

# Multi-word arithmetic

- There exists algorithms for performing arithmetic operations on numbers that are held as unevaluated sums of multiple FP numbers.
- For example, we may produce two double-precision values from one of the algorithms presented above.
- Then, if we wish to keep computing using those two numbers, we can use *multi-word arithmetic algorithms*.
- Those interested, see the Sec. 14.1 of [HFP18](#) for more details.

# Summation algorithms

- Summation of a series of FP numbers is at the core of scientific computing.
- It is required in, to name a few places, vector products, matrix vector products, matrix-matrix products, means, variances, and polynomial evaluation.
- Accumulation of values as the data is being generated occurs in ODE and PDE solvers, weight updates in machine learning, and similar.
- FP arithmetic can benefit both from changing the order of summands as well as algorithmic approaches that leverage error-free transformations.

# Summation algorithms

- Summation involves a problem of adding  $n$  values  $x_1, \dots, x_n$  in FP arithmetic.
- That is, we wish to compute  $\sum_{i=1}^n x_i$ .
- Naturally, we wish to have the best possible accuracy.
- For simplicity we will deal with nonnegative values  $x_i \geq 0$ .
- There are many techniques to perform FP summation, but the accuracy depends on the data being summed and there is no best solution generally.
- Those interested in various complex issues with different data distributions and summation, start with Chapter 4 of [ASNA02](#).

# Recursive summation

Algorithm 3.4: RecSum

```
inputs x1, ..., xn.  
s = x1  
for i = 2 to n  
    s = RN(s + xi)  
return s
```

- A straightforward solution is to read data in the order it is stored/computed and accumulate it.
- This technique is called *recursive summation*.
- On every iteration we add a relative error of up to  $u = 2^{-p}$  due to rounding in the addition operation.

- The order of operations can play an important role in reducing the final error of the computed sum.
- If we sort in decreasing order, then we will be adding increasingly small quantities to an increasingly bigger sum—this can result in more roundoff errors.

# Recursive summation

- If we instead sort in an increasing order, we will be adding increasingly large values to an increasingly large sum.
- This usually results in less shifting of the significands and therefore less rounding errors.
- Consider a problem of computing the *harmonic series*:

$$\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots$$

- This series is known as a *divergent series*, but it is also known to converge to some value in limited precision arithmetics, such as used in computers.
- We modify the problem and compute the truncated series for some number of steps  $N$ , that is, we compute  $\sum_{i=1}^N \frac{1}{N}$ .

# Recursive summation: harmonic series

- None of the limited-precision arithmetics can compute harmonic series exactly, and all of them can be said to be wrong right from the start.
- However, if we declare that double precision harmonic sum is our *reference solution*, we can compare other arithmetics to it.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     float fsum = 0;
6     double dsum = 0;
7     long int N = 1000000;
8
9     for (int i = 1; i <= N; i++) {
10         fsum += (float)1/(float)i;
11         dsum += (double)1/(double)i;
12     }
13
14     printf("fsum = %.10f, dsum = %.10f, dsum - fsum = %.10f \n",
15            fsum, dsum, dsum-fsum);
16
17 }
```

- The example code in `recursive_sum0.c` sums the harmonic series for  $N = 10^6$  in an increasing order.
- This program produces:

fsum	14.3573579788
dsum	14.3927267229
dsum-fsum	0.0353687440

# Recursive summation: harmonic series

- We can reverse the order of evaluation and sum with recursive summation from  $i = 10^6$  to  $i = 1$ , starting from the smallest addend.
- The example code is in `recursive_sum1.c`.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     float fsum = 0;
6     double dsum = 0;
7     long int N = 1000000;
8
9     for (int i = N; i > 0; i--) {
10         fsum += (float)1/(float)i;
11         dsum += (double)1/(double)i;
12     }
13
14     printf("fsum = %.10f, dsum = %.10f, dsum - fsum = %.10f \n",
15           fsum, dsum, dsum-fsum);
16 }
```

fsum	14.3926515579
dsum	14.3927267229
dsum-fsum	0.0000751649

- Notice the change in the computed single precision answer.
- The absolute error is much smaller with the values sorted in an increasing order.

# Compensated summation

Algorithm 3.5: CompSum

```
inputs  $x_1, \dots, x_n$ .  
s =  $x_1$   
t = 0  
for i = 2 to n  
    temp = RN( $x_i + t$ )  
    (s, t) = Fast2Sum(s, temp)  
end  
return s
```

The algorithm is usually attributed to W. M. Kahan.

Photo: [https://en.wikipedia.org/wiki/William\\_Kahan](https://en.wikipedia.org/wiki/William_Kahan)



- Compensated summation algorithm captures the error in each addition using Fast2Sum.
- The key idea is to use the error, induced previously, in the next step.
- If the error is positive (some quantity was removed in rounding), add it.
- If the error is negative (some quantity was added in rounding), then remove it.

# Compensated summation: harmonic series

```
12 int main() {
13     float fsum = 0;
14     double dsum = 0;
15     long int N = 1000000;
16     float t = 0;
17
18     for (int i = 1; i <= N; i++) {
19         float addend = (float)1/(float)i + t;
20         fastTwoSum(fsum, addend, &fsum, &t);
21         dsum += (double)1/(double)i;
22     }
23
24     printf("fsum = %.10f, dsum = %.10f, dsum - fsum = %.10f \n",
25           fsum, dsum, dsum-fsum);
26
27 }
```

- Example code in `compensated_sum.c`.
- It further reduces the absolute error.

fsum	14.3927268982
dsum	14.3927267229
dsum-fsum	-0.0000001753

- On line 19, we add the error from the addition in the previous step, to the next element of the series.
- On line 20, we add the sum of them to the overall sum of the series, and compute a new error.

# Cascaded summation

Algorithm 3.6: CascSum

```
inputs x1, ..., xn.  
s = x1  
t, e = 0  
for i = 2 to n  
    (s, t) = 2Sum(s, xi)  
    e = RN(e + t)  
end  
return RN(s + e)
```

- Here the core idea is to accumulate errors in a separate variable.
- Accumulated errors are added to the total sum at the end.

# Cascaded summation: harmonic series

```
15 int main() {
16     float fsum = 0;
17     double dsum = 0;
18     long int N = 1000000;
19     float t = 0;
20     float e = 0;
21
22     for (int i = 1; i <= N; i++) {
23         float addend = (float)1/(float)i;
24         twoSum(fsum, addend, &fsum, &t);
25         e += t;
26         dsum += (double)1/(double)i;
27     }
28
29     fsum += e;
30
31     printf("fsum = %.10f, dsum = %.10f, dsum - fsum = %.10f \n",
32           fsum, dsum, dsum-fsum);
33
34 }
```

- Example code in `cascaded_sum.c`.
- Worse than compensated summation in this problem.

fsum	14.3927278519
dsum	14.3927267229
dsum-fsum	-0.0000011290

- On line 24, we are adding a new element of the series to the total sum and computing the error of that addition.
- On line 25 we are adding that error into the total sum of errors.
- On line 29, when the series finishes, we add the errors to the sum.

# Stagnation

- All the presented summation algorithms are computed in the working precision, but utilize error-free transforms to improve accuracy.
- The main issue with the FP addition is a problem termed *stagnation*.
- It happens when  $\text{RN}(a + b) = a$  for some small  $b$ .
- Informally, stagnation occurs when the two numbers are so different in magnitude that the operation does not change the larger value.
- $b$  is entirely lost to rounding.
- For example, harmonic series converges due to stagnation, when an addend  $\frac{1}{i}$  becomes too small to affect the sum.
- Using mixed-precision or different summation algorithms can help in avoiding stagnation, depending on the problem.

# Stagnation

- For single precision arithmetic, stagnation in harmonic series with recursive summation in the decreasing order occurs around  $i \approx 2 \times 10^6$ .
- Example `stagnation.c` demonstrates this.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     float fsum = 0;
6     double dsum = 0;
7     long int N = 5000000;
8
9     for (int i = 1; i <= N; i++) {
10         fsum += (float)1/(float)i;
11         dsum += (double)1/(double)i;
12         if (i % 1000000 == 0)
13             printf("At iteration %d fsum = %.10f, dsum = %.10f \n",
14                   i, fsum, dsum);
15     }
16
17 }
```

# Stagnation

- Program `stagnation.c` in the examples produces

```
At iteration 1000000 fsum = 14.3573579788, dsum = 14.3927267229
At iteration 2000000 fsum = 15.3110322952, dsum = 15.0858736534
At iteration 3000000 fsum = 15.4036827087, dsum = 15.4913386782
At iteration 4000000 fsum = 15.4036827087, dsum = 15.7790207090
At iteration 5000000 fsum = 15.4036827087, dsum = 16.0021642353
```

- Notice that `fsum` stopped changing sometime after 2 000 000th iteration.
- Double precision continues to add to the overall sum.
- It has been shown in research that double precision stagnates as well after 24 days of run time on a modern processor, after iteration  $i = 2^{48}$ .

# Arbitrary-precision libraries

- One other approach to gain more accurate results is to use arbitrary precision libraries.
- The main principle is that, instead of representing the data in the working precision that the hardware supports, we can represent it in some arbitrary (might be prespecified) precision.
- Arithmetic is performed much slower, and the performance and memory utilization changes with precision.
- May or may not reuse hardware FP arithmetic, or can be based mainly on integer arithmetic.

# Arbitrary-precision libraries

Some software and libraries that include arbitrary-precision arithmetics:

- ❑ Mathematica (general purpose mathematical software),
- ❑ Maple (software for both numeric and symbolic computing),
- ❑ MATLAB Advanpix toolbox (provides fast arbitrary precision in MATLAB),
- ❑ MATLAB Symbolic Math Toolbox (another MATLAB arbitrary precision toolbox), and
- ❑ GNU MPFR Arbitrary precision library with the interface for C).

Here we use GNU MPFR for demonstrating the basic principles.

# GNU MPFR basics

- Computations are done on MPFR FP objects which represent numbers or NaN values.
- Each MPFR FP object has its own precision which is specified on initialization of an object.
- MPFR FP objects are similar to IEEE 754 since they have an exponent and a significand, but since this is arbitrary precision, they can occupy multiple registers/memory locations.
- MPFR library provides a wide array of elementary arithmetic operations as well as elementary functions, trigonometric functions, pseudo-random number generators and more, that operate on, and produce, MPFR FP objects.

# GNU MPFR example

```
1 #include <mpfr.h>
2 #include <stdio.h>
3
4 int main (void) {
5     mpfr_t num, den, res;
6     mpfr_inits2 (200, num, den, res, (mpfr_ptr) 0);
7     mpfr_set_si(num, 1, MPFR_RNDN);
8     mpfr_set_si(den, 3, MPFR_RNDN);
9
10    mpfr_div(res, num, den, MPFR_RNDN);
11
12    mpfr_printf("1/3 in 200-bit MPFR is %.100Rf \n", res);
13
14    mpfr_clears(num, den, res, (mpfr_ptr) 0);
15
16    double fres = (double)1/3;
17
18    printf("1/3 in double is      %.100f \n", fres);
19 }
```

- The number  $1/3 = 0.3333 \dots$  is a *repeating decimal* and cannot be represented in the FP arithmetic.
- We look at what the nearest value is in different computer precisions.
- Example `mpfr_example.c` computes  $1/3$  in 200-bit MPFR FP type and double precision.
- Prints out to 100 digits.

- We first include `mpfr.h` on line 1; we then initialize three MPFR type variables (`num`, `den`, `res`) with 200-bit precision (roughly 60 dec. digits) and set `num=1`, `den=3` on lines 5–8.
- On lines 10 and 12 we perform MPFR division and print out the result.
- On line 14 we clear the MPFR objects.

# GNU MPFR example

```
1 #include <mpfr.h>
2 #include <stdio.h>
3
4 int main (void) {
5     mpfr_t num, den, res;
6     mpfr_inits2 (200, num, den, res, (mpfr_ptr) 0);
7     mpfr_set_si(num, 1, MPFR_RNDN);
8     mpfr_set_si(den, 3, MPFR_RNDN);
9
10    mpfr_div(res, num, den, MPFR_RNDN);
11
12    mpfr_printf("1/3 in 200-bit MPFR is %.100RF \n", res);
13
14    mpfr_clears(num, den, res, (mpfr_ptr) 0);
15
16    double fres = (double)1/3;
17
18    printf("1/3 in double is      %.100f \n", fres);
19 }
```

- The example in `mpfr_example.c` produces the following output.
  - The approximations to  $1/3$  in 200-bit MPFR FP type and double-precision are shown. As expected, MPFR approximation has more correct digits (more 3's).

# Stochastic rounding (SR)

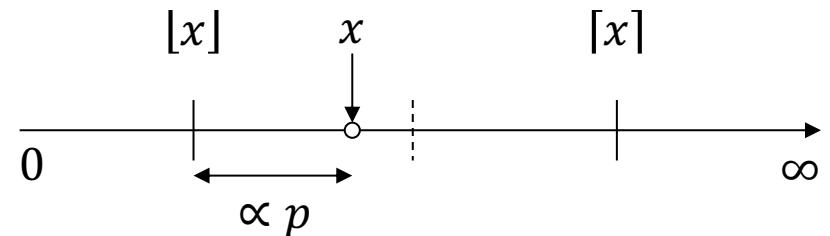
- Some latest hardware for machine learning introduced a rounding mode that does not appear in the IEEE 754 standards.
- It is usually called *stochastic rounding*.
- The main idea of stochastic rounding is to preserve some information of the bits that are thrown away in rounding.
- However, they are not stored explicitly as in error-free transformations, but make impact statistically over multiple roundings.
- Saves memory and hardware costs, since we extend precision without modifying the target precision.
- However, it has expensive rounding logic compared with other rounding modes since *(pseudo)random number generation* is required.

# Stochastic rounding (SR)

Given  $x \in \mathbb{R}$  with  $\lfloor x \rfloor \leq x \leq \lceil x \rceil$  (when  $x \notin F$  it is between the two neighbouring floats), stochastic rounding (SR) is defined as

$$\text{SR}(x) = \begin{cases} \lfloor x \rfloor & \text{with the probability } p, \\ \lceil x \rceil & \text{with the probability } 1 - p. \end{cases}$$

Mode 1	$p = 0.5$
Mode 2	$p = \frac{x - \lfloor x \rfloor}{\text{ulp}(x)}$



Here  $\text{ulp}(x)$  is a gap between  $\lfloor x \rfloor$  and  $\lceil x \rceil$ .

With mode 2,  $\mathbb{E}(\text{SR}(x)) = x$ .

# Stochastic rounding (SR)

- Consider a demonstrative example of computing, in integer arithmetic,  
$$0.25 + 0.25 + 0.25 + 0.25 = 1.$$

- Each addend has to be rounded to integer to perform the addition using an integer adder (note, in reality we would use fixed-point arith.).
- With round to nearest, we get

$$\text{RN}(0.25) + \text{RN}(0.25) + \text{RN}(0.25) + \text{RN}(0.25) = 0.$$

- Not an unexpected result since we have to round each 0.25 to the nearest integer, 0.
- With stochastic rounding mode 2 we most likely get

$$\text{SR}(0.25) + \text{SR}(0.25) + \text{SR}(0.25) + \text{SR}(0.25) = 1.$$

- The probability of rounding 0.25 to 1 is  $p = \frac{1}{4}$ , whereas rounding to 0 it is  $1 - p = \frac{3}{4}$ . Therefore, one out of 4 roundings above produces 1.