Precision Loss in Floating Point Numbers

Pedram Pasandide

Introduction

Floating-point numbers in computers are an approximation of real numbers and are limited by the precision of the representation. The most common sources of precision loss are rounding errors and truncation errors, both of which occur because a finite number of binary digits (bits) must be used to represent an infinite or very large set of real numbers. This limitation becomes evident when performing arithmetic operations, especially when working with numbers that require more precision than the computer can provide.

1 Finite Precision (Rounding Errors)

Floating-point numbers have a limited number of bits, so many real numbers cannot be represented exactly, leading to rounding errors.

```c
int main() {
 float a = 0.1f;
 double b = 0.1;

 printf("Float a  = %.20f\n", a);
 printf("Double b = %.20lf\n", b);

 // The addition of these numbers obviously can create rounding
    errors:
 printf("a+b = %.20lf\n", a+b);

 return 0;
}
```

The value 0.1 cannot be represented exactly in binary, leading to small precision errors. Output:

```
Float a  = 0.10000000149011611938
Double b = 0.10000000000000000555
a+b = 0.20000000149011612494
```

Solution: The only solution here is to use a higher precision such as infinite precision which it can be computationally costly.

2 Accumulation of Errors

When performing many arithmetic operations, small rounding errors accumulate.

```c
int main() {
 float sum = 0.0;
 for (int i = 0; i < 1000000; i++) {
  // Adding a small value repeatedly
  // (the small value already has round-off):
  sum += 0.0001;
 }

 printf("Expected: 100.0\n");
 printf("Actual   : %.10f\n", sum);

 return 0;
}
```

The accumulated rounding error results in a loss of precision. Output:

```
Expected: 100.0
Actual   : 99.3273086548
```

Solution:

- Rearrange calculations to minimize precision loss (e.g., sum small numbers first).
- Use Kahan Summation Algorithm for accurate summation.
- Use compensated arithmetic methods, such as pairwise summation.

3 Catastrophic Cancellation

When subtracting two nearly equal floating-point numbers, significant digits can be lost.

```c
int main() {
 double a = 1.0000001;
 double b = 1.0000000;
 double result = (a - b);

 // 1.0000001 - 1.0000000 = 0.0000001 OR = 1e-7
 double Expected = 1e-7;

 printf("With .15lf:\n");
 printf("Expected: %.15lf\n", Expected);
 printf("Actual   : %.15lf\n", result);

 // However, with 15 decimals the difference cannot be seen. So:
 printf("\nWith .20lf:\n");
 printf("Expected: %.20lf\n", Expected);
 printf("Actual   : %.20lf\n", result);

 // But how many decimals are accurate to be compared?
 // We need to use floating point representation.
 // Then always 15 decimals are accurate
 printf("\nWith .15e:\n");
 printf("Expected: %.15e\n", Expected);
 printf("Actual   : %.15e\n", result);

 // Now the difference is easy to see

 return 0;
}
```

Loss of precision occurs because the difference is much smaller than the individual numbers, making rounding errors more pronounced. Output:

```
With .15lf:
Expected: 0.000000100000000
Actual  : 0.000000100000000
```

```
With .20lf:
Expected: 0.00000010000000000000
Actual  : 0.00000010000000005839

With .15e:
Expected: 1.000000000000000e-07
Actual  : 1.000000000583867e-07
```

Solution:

- Use algebraic transformations to avoid direct subtraction. For example, instead of using $\sqrt{a} - \sqrt{a}$, use:

  ```
  // More stable transformation
  double result = (a - b) / (sqrt(a) + sqrt(b));
  ```

  Note: $\sqrt{a} - \sqrt{b} = \sqrt{a} - \sqrt{b} \times \dfrac{\sqrt{a} + \sqrt{b}}{\sqrt{a} + \sqrt{b}} = \dfrac{a - b}{\sqrt{a} + \sqrt{b}}$

- Using a higher precision from `float` to `double` works here. However, the same catastrophic cancellation can still happen in a higher precision.

4 Large and Small Values (Underflow & Overflow)

Very large or very small values may exceed the range of floating-point representation.

```
#include <float.h>

int main() {
 float large = FLT_MAX;
 float small = FLT_MIN;

 printf("Large: %e\n", large);
 printf("Small: %e\n", small);

 float overflow = large * 2.0f;
 float underflow = small / 2.0f;

 printf("Overflow: %e\n", overflow);  // Infinity
 printf("Underflow: %e\n", underflow); // 0
```

```
  return 0;
}
```

Overflow results in infinity, and underflow rounds to zero. Output:

```
Large: 3.402823e+38
Small: 1.175494e-38
Overflow: inf
Underflow: 0.000000e+00
```

Overflow results in infinity, and underflow rounds to zero. In case `Underflow:  5.877472e-39` is printed instead of zero, you can read Appendix.

Solution:

- Use logarithms for multiplication and division of very large or small numbers. For example, instead of:

  ```
  double result = a * b * c;
  ```

  use:

  ```
  double result = exp(log(a) + log(b) + log(c));
  ```

  This prevents overflow by working in log-space.

  Note: Using `log` and `exp` can introduce round-off and truncation errors. Finding a balance in between is what engineers do!

- Use `double` instead of `float` if working with extreme ranges.
- Check for `FLT_MAX` and `FLT_MIN` before performing calculations.

5 Floating-Point Division (Non-Terminating Decimals)

Some simple fractions (like 1/3) cannot be represented exactly in binary.

```
int main() {
  float a = 1.0f / 3.0f;
  double b = 1.0 / 3.0;
```

```
    printf("Float a  = %.20f\n", a);
    printf("Double b = %.20lf\n", b);

    return 0;
}
```

Output:

```
Float a  = 0.33333334326744079590
Double b = 0.33333333333333331483
```

The fraction 1/3 is a repeating decimal in base 10 and cannot be stored exactly in floating-point.

Solution:

- Use rational approximations when possible.
- Using a higher precision.
- Use symbolic computation (fractions) if the exact value is necessary. For example:

  ```
  int numerator = 1, denominator = 3;
  printf("Exact Fraction: %d/%d\n", numerator, denominator);
  ```

Conclusion

We've explored the fundamental sources of precision loss in floating-point numbers, including rounding errors and limitations in the representation of real numbers. Through examples, we've demonstrated how even numbers that appear to be simple can lead to small but significant discrepancies when represented in computer memory. We've also discussed the role of the `float` and `double` types in handling numerical precision and the importance of understanding these concepts for developing accurate numerical computations in software.

However, this is just the beginning. Floating-point arithmetic is a deep and complex topic, and there are many other areas we have not covered, such as floating-point exceptions, error analysis in numerical methods, precision vs. performance trade-offs, and the nuances of different floating-point formats (such as single, double, and extended precision). Additionally, advanced techniques like arbitrary-precision arithmetic and specialized hardware optimizations for floating-point operations are topics worth exploring for those interested in high-precision calculations or developing software for scientific computing.

If you find these topics intriguing, we encourage you to continue studying them, as a deeper understanding of floating-point arithmetic will help you write more accurate, efficient, and reliable programs. There are many resources available to dive deeper into the theoretical and practical aspects of floating-point precision and numerical computing.

Appendix

The smallest normalized `float` is 1.175494e-38, and any number smaller than the number is supposed to be rounded to zero.

Floating-point computations involving extremely small numbers (subnormal or denormal numbers) can lead to unexpected results due to how modern processors handle underflow. Consider the following two C programs:

Example 1: Underflow in Input Stage (`input_stage.c`):

```
int main() {
 float x = 1.175494e-38 / 2.0f;
 // Should underflow to zero
 printf("Underflowed value: %e\n", x);
}
```

In this code, the value 1.175494e-38 / 2.0f is intended to underflow and produce a denormal value. However, instead of producing 0.000000e+00 (which would be the expected result), the output is 5.877470e-39. This happens because denormal numbers are treated as valid, non-zero values, even though they are very close to zero.

Example 2: Underflow in Computation Stage (`computation_stage.c`):

```
int main() {
 float x = 1.175494e-38; // Smallest normal number (FLT_MIN)
 float y = 2.0f;         // Devide by to create an underflow
 // float underflow in computation stage:
 float result = x / y;
 printf("Result (0.000000e+00 Expected but): %e\n", result);
}
```

Similarly, in this code, dividing 1.175494e-38 by 2.0f results in an underflow, which leads to a denormal number instead of the expected zero. The result again is 5.877470e-39, rather than 0.000000e+00.

To address the unexpected results, we first need to understand Denormals-Are-Zero (DAZ) and Flush-To-Zero (FTZ) modes. These two modes are hardware features that control how very small numbers (denormals) are handled during computation.

Denormals-Are-Zero (DAZ)

Denormals are numbers smaller than the smallest normal floating-point number. They can be represented, but at the cost of performance and precision. DAZ mode forces all denormal input values to be treated as zero before they are used in computations. This is useful when you want to prevent denormal values from affecting your calculations, which can slow down processing.

Flush-To-Zero (FTZ)

FTZ mode is used to flush any results that underflow (i.e., become too small to be represented normally) to zero. FTZ affects the computation stage—if an operation results in a number that is too small to be represented normally, it is flushed to zero. However, FTZ does not affect input denormals, which are still passed into computations unless DAZ is enabled.

How to Check and Enable DAZ and FTZ

To check if DAZ and FTZ are enabled, we use the following functions from the SSE3 instruction set in C:

```c
#include <stdio.h>
#include <pmmintrin.h>  // SSE3 Instructions (for DAZ)

int main() {
 if (_MM_GET_DENORMALS_ZERO_MODE() == _MM_DENORMALS_ZERO_ON) {
  printf("Denormals-Are-Zero (DAZ) is ENABLED!\n");
 } else {
  printf("Denormals-Are-Zero (DAZ) is DISABLED!\n");
 }

 if (_MM_GET_FLUSH_ZERO_MODE() == _MM_FLUSH_ZERO_ON) {
  printf("Flush-to-Zero (FTZ) is ENABLED!\n");
 } else {
  printf("Flush-to-Zero (FTZ) is DISABLED!\n");
 }
```

```
    return 0;
  }
```

If you have the similar issue, you should see:

```
Denormals-Are-Zero (DAZ) is DISABLED!
Flush-to-Zero (FTZ) is DISABLED!
```

To enable DAZ:

```
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
```

To enable FTZ:

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
```

Relation to the Issue

In `input_stage.c`, enabling DAZ will cause any denormal **input** values to be set to zero before they are used in computations. This is why when DAZ is enabled, the result will correctly be 0.000000e+00. However, enabling FTZ does not affect input values, so it does **not** fix the issue in this case.

In `computation_stage.c`, enabling FTZ or DAZ will both result in the underflowed result being flushed to zero since FTZ affects the output of **computations**, and DAZ affects input denormals. Thus, when either FTZ **or** DAZ is enabled, the expected result 0.000000e+00 will be achieved.

Getting the Expected Result

For `input_stage.c` (with DAZ ON): In this case, the input value is very small, and enabling DAZ ensures that the denormal value is treated as zero before the computation.

```
#include <stdio.h>
#include <pmmintrin.h>

int main() {
  // Enable DAZ (Denormals-Are-Zero) to set input denormals to
     zero
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  float x = 1.175494e-38 / 2.0f;  // Smallest normal number (
     FLT_MIN)/2.0f
  printf("DAZ Enabled:\n");
  printf("Underflowed value Input stage: %e\n", x);  // Expected
     result: 0.000000e+00
  return 0;
}
```

Expected Output:

```
DAZ Enabled:
Underflowed value Input stage: 0.000000e+00
```

For `computation_stage.c` (with FTZ ON, it also works with DAZ ON): In this case, enabling FTZ ensures that the underflowed result from the division is flushed to zero.

```
#include <stdio.h>
#include <pmmintrin.h>

int main() {
  // Enable FTZ (Flush-To-Zero) to flush underflowed results to
     zero
  _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  float x = 1.175494e-38;  // Smallest normal number (FLT_MIN)
  float y = 2.0f;          // Divide by to create an underflow
  // float underflow in computation stage:
  float result = x / y;
  printf("FTZ Enabled:\n");
  printf("Underflowed value Computation stage: %e\n", result);  //
      Expected result: 0.000000e+00
  return 0;
}
```

Expected Output:

```
FTZ Enabled:
Underflowed value Computation stage: 0.000000e+00
```

By enabling DAZ, we ensure that input denormal values are treated as zero, which resolves the issue in `input_stage.c`. In contrast, enabling FTZ handles computation underflows by flushing them to zero, fixing the issue in `computation_stage.c`. Both FTZ and DAZ are critical for managing floating-point precision issues related to very small numbers in computations.