Computational Physics.
Problem of “0.1 + 0.2”

30407 김종원

(0.1 + 0.2)가 0.3과 동치라는 논리가 거짓으로 도출되는 이유에 대해 서술하겠다.

Python에서는 실수 자료형은 8바이트, 즉 64비트로, 또한 IEEE 754 방식의 부동소수점 방식을 사용하여 소수를 포함한 실수를 저장한다.

여기에서 1바이트는 8비트를 뜻하고, 각 비트에는 0과 1이 들어갈 수 있다.

1. 배정밀도와 단정밀도

배정밀도와 단정밀도는 소수 표현 방식의 정밀도를 위해 어느 만큼의 비트를 할당하여 연산할 것 인지를 나타내는 수치이다. 단정밀도(single precision)는 32비트를 사용하는 것이고, 배정밀도(double precision)는 64비트를 사용하는 것을 이야기한다.

2. 소수를 표현하는 방식

위에서 배정밀도와 단정밀도에 대해 언급하였다. 이때, 소수를 나타내기 위해서는 할당된 정밀도에 따라 비트 수를 할당해주어야 한다. 이때, 32/64 비트 고정소수점 방식에서는, 부호 비트에 1비트를 할당한 후, 나머지 정수부 n비트, 소수부 m비트의 합인 n+m이 32/64 비트가 되도록 할당한다. 이때 정수부 비트와 소수부 비트의 경계를 소수점으로 생각하여 소수를 저장하는 방식이 고정소수점이다. 그리고, 부동소수점의 비트 할당은, 32비트의 경우 부호비트 1비트, 지수부 8비트, 가수부 23비트로 할당하며 64비트(파이썬에서 사용되는 실수형)의 경우에는 부호비트 1비트, 지수부 11비트, 가수부 52비트로 할당되어 연산된다.

3. 이진수 연산

컴퓨터 상에 데이터를 저장하기 위해서는 이진수의 표현으로 수를 바꾸어 저장해야 한다. 10진수 정수의 이진수 변환은 널리 알려져 있듯 수를 2로 반복적으로 나눠 1또는 0이 나올 때까지 반복한 이후, 각 단계의 나머지를 역순으로 정렬한 값이 이진수가 된다. 하지만 소수를 포함한 10진수 실수들을 저장하기 위해서는 소수점을 기준으로 정수부와 소수부를 나눈 후 각각 이진수로 변환한 뒤 합쳐주어야 한다. 예를 들어 소수의 이진수 변환의 대표적인 예시인 13.625를 변환해 보자.

13.625를 이진수로 변환하면, 먼저 정수부 13을 변환해야 한다. 13을 2로 나누면서 나머지를 기록하면, 13 ÷ 2 = 6 (나머지 1), 6 ÷ 2 = 3 (나머지 0), 3 ÷ 2 = 1 (나머지 1), 1 ÷ 2 = 0 (나머지 1)이 된다. 이를 역순으로 정렬하면 정수부의 이진수 표현은 1101이 된다.

이제 소수부 0.625를 이진수로 변환한다. 소수부는 2를 곱하면서 정수 부분을 따로 정렬하는 방식으로 변환할 수 있다. 0.625 × 2 = 1.25에서 정수 부분 1을 기록하고, 0.25를 다시 2로 곱한다. 0.25 × 2 = 0.5에서 정수 부분 0을 기록하고, 0.5를 다시 2로 곱한다. 0.5 × 2 = 1.0에서 정수 부분 1을

기록하며, 소수부가 0이 되므로 변환이 종료된다

따라서 소수부의 이진수 표현은 0.101이 되고, 정수부와 소수부를 합쳐 최종적으로 13.625의 이진수 표현은 1101.101_2이 된다.

4. 고정소수점(Fixed-Point Number Representation)

기존에는 고정소수점이라는 방식으로 소수를 저장하였다. 앞서 설명한 소수의 이진수 변환 방식과 함께 이 고정 소수점 방식을 사용할 수 있다. (2)에서 언급했듯, 고정소수점은 정수부와 소수부의 경계를 소수점으로 생각하여 각 비트에 수를 할당해주는 방식이다. (3)에서 연산한 13.625를 고정소수점 방식으로 저장한다고 가정해보자.

위 표에 따라 고정소수점을 위한 비트 수를 할당하였다고 가정해보자. 주황색 부분은 부호비트, 녹색은 정수부, 자색은 소수부이다. 이때 13.625를 이진수로 나타내면 (3)에 따라 1101.101_2 이므로, 녹색과 자색의 경계를 소수점으로 생각하고 이진수를 할당하면 다음과 같다.

위와 같이 소수점을 기준으로 양 옆으로 저장을 한 이후, 나머지 비트는 모두 0으로 채워주는 방식이다. 부호비트는 양수인 경우에는 0, 음수인 경우에는 1이다.

결론적으로, 고정소수점 방식으로 수를 저장하였다.

4.1 고정소수점 방식의 한계

하지만, 이러한 고정소수점 방식은 수를 표현하는 데 여러 가지 한계를 가진다. 먼저, 가장 큰 문제는 표현 범위의 제한이다. 고정소수점 방식에서는 정수부와 소수부의 비트 수를 미리 정해두기 때문에, 표현할 수 있는 값의 범위가 고정된다. 예를 들어, 16.16 형식(32비트를 16/16 형태로 나눈 형태로, 부호비트 1비트 + 정수부 15, 그리고 소수부 16으로 나누어 대칭을 이루는 형태)에서는 정수부가 16비트이므로 표현할 수 있는 최대 정수 값이 제한되며, 소수부가 16비트이므로 소수 부분의 정밀도도 제한된다. 따라서 너무 큰 수를 표현하려 하면 오버 플로우가 발생하고, 너무 작은 수를 표현하려 하면 언더 플로우가 발생할 수 있다.

또한, 연산의 유연성이 부족하다는 문제도 있다. 부동소수점 방식과 달리, 고정소수점 방식에서는 소수점의 위치가 고정되어 있으므로, 서로 다른 범위의 숫자를 연산할 때 매우 불편하다. 예를 들어, 큰 정수를 다루기 위해 정수부 비트를 많이 할당하면 소수부의 정밀도가 떨어지고, 반대로 소수부의 정밀도를 높이면 표현할 수 있는 정수 범위가 줄어든다. 이로 인해, 다양한 크기의 수를 처리해야 하는 과학 연산이나 금융 계산 등에서는 고정소수점 방식이 적합하지 않을 수 있다.

마지막으로, 메모리 및 연산 효율성 문제도 있다. 고정소수점 연산은 특정한 비트 수를 기준으로 덧셈과 뺄셈을 수행하는 데는 상대적으로 효율적이지만, 곱셈과 나눗셈 연산에서는 비효율적일 수 있다. 특히, 소수부를 포함한 곱셈 연산을 수행할 경우 소수점의 위치를 고려해야 하므로 추가적인 연산이 필요하며, 결과값을 다시 고정된 비트 크기에 억지로 할당하거나 크기에 맞게 할당해야 하는 번거로움이 있다.

총체적인 관점에서, 고정소수점 방식은 연산 속도가 빠르고 하드웨어와 연계된 구현이 간단하다는 장점이 있지만, 표현 범위가 제한적이고 다양한 크기의 수를 유연하게 다루기 어렵다는 단점이 있다. 이러한 한계로 인해, 보다 넓은 범위와 높은 정밀도를 필요로 하는 계산에서는 부동소수점 방식이 선호된다.

5. 부동소수점(Floating-Point Number Representation)

앞서 살펴본 고정소수점의 한계를 극복하고자 부동소수점이라는 새로운 소수 저장을 위한 방법론이 대두되었다. 역사적으로 다양한 부동소수점 방식의 형태가 존재했는데, 현재에는 전기전자공학자협회(Institute of Electrical and Electronics Engineers, 일명 IEEE)에서 고안한 부동소수점 방식인 IEEE 754 방식이 표준으로 사용되고 있다. 파이썬을 비롯한 여러 언어에서도 이 방식을 따라 소수를 저장하고 있다.

5.1 부동소수점의 새로운 소수 표현 방식

고정소수점에서 사용했던 소수 표현 방식과 달리, 부동소수점에서는 다른 표현방식을 사용한다.

(부호) x (가수) x (밑수)^(지수)

위와 같은 방식의 표현방식을 사용한다. 정수부 소수부가 아닌 유효숫자를 사용한 곱셈 형태로 표현한다. 예를 하나 들어 살펴보자. 앞서 위에서 사용한 수인 13.625 라는 10진수 실수를 다시 가져와서 사용하자. 10진수와 2진수 두가지 표현을 살펴보자.

10진수 – 13.625 x 10^1

2진수 – 1101.101 x 2^1

위와 같이 각각의 표현을 살펴보면, 가수와 n진법에 해당하는 n의 m제곱 꼴로 나타나는 표현이다. 이때, 가수의 소수점을 왼쪽으로 한 칸 옮기면 m에 해당하는 수가 1 늘어나게 되는 방식이다.

10진수 – 1.3625 x 10^2

2진수 – 110.1101 x 2^2

앞서 설명한 내용대로 소수점이 왼쪽으로 한 칸씩 옮겨졌을 때, 밑수의 지수가 1씩 늘어난 모습이다.

5.2 정규화 과정

부동소수점의 정규화란, 가수의 첫번째 자리가 밑수보다 작은 한자리 자연수로 바꾸는 것을 의미한다. 위에서 사용한 예시를 바탕으로 정규화를 진행하면 다음과 같다.

10진수 – 1.3625 x 10^2

2진수 – 1.101101 x 2^4

이는 부동소수점 방식에서 유효 숫자의 표현을 최대한 효율적으로 사용하기 위해서이다. 정규화를 수행하면 가수의 첫 번째 자리가 항상 1이 되므로, 이를 저장할 때 생략할 수 있어 메모리 절약과 연산 최적화가 가능하다. 예를 들어, IEEE 754 표준에서는 이러한 정규화를 이용해 가수의 첫 번째 비트(숨겨진 비트, Hidden Bit)를 저장하지 않고도 값의 정확성을 유지할 수 있다.

또한, 정규화를 통해 일관된 표현 방식을 유지할 수 있어, 숫자의 비교 및 연산이 훨씬 간편해진다. 만약 정규화를 하지 않는다면, 같은 값을 여러 가지 방식으로 표현할 수 있어 연산 전에 불필요한 변환 과정이

필요하게 된다. 하지만 정규화된 상태에서는 항상 동일한 형식으로 저장되므로, 연산이 단순해지고 계산 과정에서 발생할 수 있는 오류도 줄어든다.

5.3 IEEE 754 표준

타입	부호	지수부	가수부	총 비트수
Half precision	1	5	10	16
Single precision	1	8	23	32
Double precision	1	11	52	64
X86 extended precision	1	15	64	80
Quad precision	1	15	112	128

IEEE 754는 IEEE에서 정의한 컴퓨터의 부동소수점 표현 방식에 대한 표준이다. 이 표준은 1985년에 제정되었으며, 현재 대부분의 컴퓨터 시스템에서 사용되고 있다. IEEE 754에서 정의하는 부동소수점 형식에는 단정밀도(32비트), 배정밀도(64비트), 확장 정밀도(80비트), 쿼드 정밀도(128비트) 등이 있으며, 각 형식은 부호 비트, 지수부, 가수부로 구성된다.

일반적으로 32비트 부동소수점은 float 타입으로, 64비트 부동소수점은 double 타입으로 표현되는데, 이는 IEEE 754에서 64비트 형식을 배정밀도(double precision)로 정의한 데서 유래한다 #재차 언급하지만, Python에서의 실수는 64비트로 표현됨. 일부 프로그래밍 언어에서는 32비트 부동소수점을 single로 표현하기도 한다.

부동소수점 방식에서는 정규화 과정을 거쳐 일관된 형식으로 값을 표현하며, 가수부의 크기에 따라 정밀도가 결정된다. 예를 들어, 123456.7890과 같은 실수를 가수부 5자리로 표현해야 한다면, 정규화를 통해 1.2345×10^5의 형태로 변환되며, 가수부에서 벗어난 값은 버려진다. 따라서 가수부가 클수록 값의 손실이 적어지고, 더 높은 정밀도를 유지할 수 있다. 이러한 이유로, 32비트 부동소수점보다 가수부가 더 큰 64비트 부동소수점을 배정밀도(double precision)라고 부르게 되었다.

5.4 부동 소수점의 5가지 반올림 방식

다음은, 표현의 한계로 인해 잘려나가는 부분을 처리하는 과정에서, 어떤 기준으로 반올림 할 것이지 IEEE 754에서 정의한 5가지 방법이다.

- 짝수로 반올림(round to nearest, ties to even)
- 큰 절대값으로 반올림(round to nearest, ties away from zero)
- 올림(round toward +∞)
- 버림(round toward -∞)
- 절삭(round toward zero)

1. Round to Nearest, Ties to Even (가장 가까운 정수로 반올림, 동점 시 짝수 선택)
   소수 부분이 정확히 0.5일 때, 앞자리 숫자가 짝수면 그대로 유지, 홀수면 한 칸 증가
   예) 4.5 → 4 (짝수 유지), 5.5 → 6 (홀수 증가)

2. Round to Nearest, Ties Away from Zero (가장 가까운 정수로 반올림, 동점 시 0에서 멀어지는 방향)
   소수 부분이 0.5이면 항상 더 큰 정수로 반올림
   예) 4.5 → 5, 5.5 → 6

3. Round Toward +∞ (양의 무한대로 반올림)
   항상 큰 정수 방향으로 반올림
   예) 3.2 → 4, 5.7 → 6

4. Round Toward -∞ (음의 무한대로 반올림)
   항상 작은 정수 방향으로 반올림
   예) 3.7 → 3, -2.8 → -3

5. Round Toward Zero (0을 향해 반올림)
   소수 부분을 버리고 정수 부분만 유지
   예) 3.7 → 3, -2.3 → -2

반올림 모드는 프로그래밍 언어 또는 하드웨어마다 일부 상이할 수도 있다. 이는 연산 결과에 영향을 미치고, IEEE 754에서는 round to nearest, ties to even를 디폴트 값으로 사용하고 있다.

5.5 IEEE 754 부동소수점 할당

이제, 본격적으로 IEEE 754 방식의 부동소수점 할당이 어떻게 이루어지는지 살펴보자. 우리가 전산물리 수업에서 사용하는 언어는 파이썬 이므로, 파이썬 실수 자료형의 비트인 64비트, 즉 Double precision 기준으로 설명하겠다.

64개의 비트를 부호비트 1개, 지수부 11개, 가수부 52개로 나누어준다. (워드프로세서 표 칸수 제한 상의 이유로 대략적인 비율만 나타내었다.)

먼저, 13.625를 2진수로 변환하면 1101.101_2가 된다. 이를 IEEE 754의 64비트 배정밀도(double precision) 형식으로 변환하기 위해 정규화 과정을 거치면 1.101101×2^4의 형태로 나타낼 수 있다. 이때, 부동소수점 표현에서 부호, 지수, 가수의 세 부분으로 나누어 저장하게 된다.

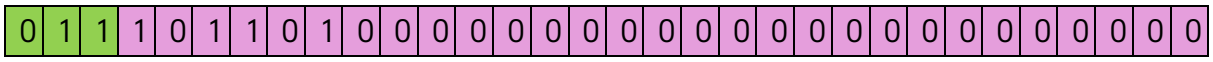
부호 비트는 13.625가 양수이므로 0이 된다. 그리고 IEEE 754 부동소수점 방식에서는 지수부를 Bias(편향) 값을 적용해 저장한다. 이 형식에서는 Bias의 값이 1023이다. 이 Bias 값을 밑수의 지수에 더해준다. 1023에 4를 더하면 1027이다. 이제 이 1027이라는 값을 2진수로 변환하면,

10000000011_2이고, 이 값을 지수부의 11칸에 왼쪽부터 차례대로 할당해주는 것이다. 11칸의 지수부 비트를 모두 채우지 못할 경우, 남는 칸은 0으로 채워주며, 만약 지수부에 대입할 값이 11칸을 초과한다면 (5.4)에서 설명한 반올림 규칙을 적용하여 11칸의 비트에 맞추어 저장해준다.

64비트 중 부호비트 및 지수부를 포함한 앞부분

앞서 서술한 결과는 위 그림과 같이 나타난다.

다음으로, 가수부는 정규화된 수에서 첫 번째 1을 생략하고 소수점 이하의 비트만 저장하게 되는데, 13.625의 경우 1.101101×2^4에서 소수점 뒤 숫자는 101101이므로 이를 가수부에 왼쪽부터 차례로 대입하고, 나머지의 경우 지수부와 동일하게 52비트로 맞추기 위해 뒤에 0을 채워준다.



64비트 중 지수부의 일부 및 가수부의 앞부분

결과적으로 13.625를 IEEE 754 64비트 형식으로 표현하면

0 | 부호비트

10000000011 | 지수부

1011010000000000000000000000000000000000000000000000 | 가수부

가 된다.

5.6 문제: 0.1 + 0.2 != 0.3

IEEE 754 표준 부동소수점 방식을 이해했으니, 0.1 + 0.2가 0.3과 동치라는 논리에 오류가 있는 이유를 살펴볼 수 있다.

파이썬에서 0.1 + 0.2를 계산하면 0.30000000000000004가 나온다. 이 문제의 원인으로는 부동소수점 연산의 정밀도 문제 때문이라고 알려져 있는데, 이를 명확하게 살펴보자.

위에서도 설명했듯이, 파이썬의 실수 자료형은 IEEE 754 표준 64비트 부동소수점을 따른다. 이때 십진수 0.1과 0.2는 2진수로 정확히 표현할 수 없다. 이제 이 과정을 서술하겠다. 앞서 언급한 소수의 이진수 변환 과정에 따르면,

0.1)

0.1 x 2 =	0.2	0
0.2 x 2 =	0.4	0
0.4 x 2 =	0.8	0
0.8 x 2 =	1.6	1
0.6 x 2 =	1.2	1
0.2 x 2 =	0.4	0

… 이후 0011이 무한히 반복되는 구조를 띄고,

$0.1 = 0.0001100110011001100110011001100110011..._2$ 로 표현할 수 있다.

이제 이를 정규화 하면, 가수의 첫번째 자리가 밑수보다 작은 한자리 자연수여야 하므로, $1.1001100110011001100110011001100110011\ldots_2 \times 2^{-4}$으로 표현된다. 따라서 지수부는 밑의 지수 + Bias 이므로 -4 + 1023 = 1019이며, 가수부는 소수점 밑 10011001100…가 들어가게 되는데, 마지막 비트를 반올림하므로, 최종적으로는 다음과 같은 형태로 들어가게 된다.

0.1 – 부동소수점
0 | 부호비트
01111111011 | 지수부
10011001100110011001100110011001100110011001100110011010 | 가수부

이렇게 이진수와 부동소수점 방식으로 표현한 0.1을 위 과정의 역순으로 다시 10진수로 변환하면, 약 10진수로 변환하면 약 0.10000000000000000555 정도의 값이 나오게 된다.

0.2 또한 마찬가지이다.

0.2)

0.2 x 2 =	0.4	0
0.4 x 2 =	0.8	0
0.8 x 2 =	1.6	1
0.6 x 2 =	1.2	1
0.2 x 2 =	0.4	0
0.4 x 2 =	0.8	0

… 이후 0.1과 마찬가지로 0011이 무한히 반복되는 구조를 띄고,
$0.2 = 0.0001100110011001100110011001100110011\ldots_2$ 로 표현할 수 있다.
마찬가지로 정규화 하면, $1.1001100110011001100110011001100110011\ldots_2 \times 2^{-3}$ 이고,
지수 + Bias = -3 + 1023 = 1020이고, 최종적으로는 아래와 같이 나타낼 수 있다.

0.2 – 부동소수점
0 | 부호비트
01111111100 | 지수부
10011001100110011001100110011001100110011001100110011010 | 가수부

이를 10진수로 변환하면 0.20000000000000000111정도의 값이 나오게 된다.

따라서 $0.1 \approx 0.10000000000000000555$, $0.2 \approx 0.20000000000000000111$ 이고,
$0.10000000000000000555 + 0.20000000000000000111 = 0.30000000000000001665$이 도출되게 된다.

이때, 계산의 결과를 다시 부동소수점 방식으로 저장해야 한다.

0.30000000000000001665을 다시 이진수로 나타내면, 0.300000000000000166510 = 0. 1001100110011001100110011001100110011001100110011001₂ 와 같은 꼴로 나타나게 된다, 이때, 0.30000000000000001665는 0.3보다 아주 약간 크므로, 0.3처럼 0011의 반복 끝에 추가적인 비트인 011이 붙게 된다. 이와 같이 변환 과정에서 미세한 오차가 나타나게 된다. 따라서, 이를 이진수로 나타내면,

0.30000000000000001665 - 부동소수점
0 | 부호비트
01111111101 | 지수부
1001100110011001100110011001100110011001100110011001 | 가수부

이와 같이 나타나게 되고, 이를 역순으로 하여 다시 십진수로 변환한 결과가 0.30000000000000004로 나타나게 되는 것이다.

즉 0.1 + 0.2는 부동소수점 표현 방식 중 가수부의 비트 수 한계의 이유로, 각각 근사하지만 조금의 오차가 있는 값으로 연산되고, 결론적으로는 0.3에서 조금의 오차가 있는 수인 0.30000000000000004로 나타나게 되는 것이다.