# 2025 年度 科学技術計算

学生番号 36714143
情報工学科知能情報分野
和田小百合

2025 年 11 月 4 日

## 目次

# 1　課題 02-2

## 1.1　課題の内容

コンピュータで計算をするときは 2 進数に変換する際に誤差を生んでしまうことがあります。これについて bit 表現から解くことによってなぜ誤差が含まれてしまうか説明します。

## 1.2　IEEE 754 binary64 とは

IEEE 754 binary64 とは、2 進数浮動小数点形式の 1 種です
構成としては、

- 一般形：$(-1)^{\text{sign}}(d_0.d_1d_2d_3\cdots d_{52})_2 \times 2^{n\ -\ 1023}$
  - sign：符号 1 ビット（正負を表す）
  - $(d_0.d_1d_2d_3\cdots d_{52})_2$：仮数部 52 ビット＋正規化表現の仮定された 1 ビットで計 53 ビットの 2 進数
    - 正規化表現の場合，$d_0 = 1$ が仮定される（格納されない）
    - $n = (n_{11}\cdots n_2n_1n_0)_2$：指数部 11 ビット（$1 \leq n \leq 2046$，バイアスは 1023）
    - $n$ は符号なし整数で表現され，指数部の範囲は $-1022 \leq n - 1023 \leq 1023$ となる。

## 1.3　コードの説明

```
a = 18.0
b = 2.0**57
c = (a + b) - b
print(c)
```

Listing 1　課題 02-2

上のコードは本来 c = 18.0 になるはずですが、実際には 32.0 が出力されます。この理由を浮動小数点数の演算手順をビット単位で追跡することでなぜ違う値が出力されたを説明します。まず、a = 18.0, b = 2.0**57 を IEEE754 binary64 のビット表現に変換し、それぞれの符号・指数・仮数を確認する。
また、a, b の符号・指数・仮数部分を表示するために今回の講義の教科書部分にあるコードを用い、表示させました。

```
def float64_to_binary(f: float | np.float64,separate: bool = False) -> str |
    Tuple[str, str, str]:
    """convert float64 to 64 bits as string

        Args:
        f (float | np.float64): IEEE754 binary64 floating point number
        separate (bool, optional): return tuple of (sign bit, exponent bits,
            mantissa bits) if True,
            return 64 bits if False. Defaults to False.

    Returns:
        str or Tuple of str: bit string of f
    """
    assert isinstance(f, float)
    from struct import pack, unpack
```

```
14
15      # see https://note.nkmk.me/python-float-hex/
16      s = format(unpack('>Q', pack('>d', f))[0], "064b")
17
18      if separate:
19          return s[0:1], s[1:12], s[12:]
20      else:
21          return s
22
23  def sign_str_to_str(s: str) -> str:
24      """sign bit to +/-
25
26      Args:
27          s (str): "0" or "1"
28
29      Returns:
30          str: "+" if s is "0" else "-" (if s is "1")
31      """
32      assert isinstance(s, str)
33      assert len(s) == 1
34      assert s in ["0", "1"]
35
36      return "+" if s == "0" else "-"
37
38  def exponent_str_to_int(
39          e: str,
40          len_e: int = 11,
41          bias: int = 1023
42      ) -> Tuple[int, bool]:
43      """exponent string to exponent integer
44
45      Args:
46          e (str): exponent string consiting of of "0" and "1"
47          len_e (int, optional): the number of bits for the exponent. Defaults to
                11.
48          bias (int, optional): bias to exponent. Defaults to 1023.
49
50      Returns:
51          Tuple[int, bool]: exponent in decimal integer, and True if normalized
               number else False
52      """
53      assert isinstance(e, str)
54      assert len(e) == len_e
55      for bit in e:
56          assert bit in ["0", "1"]
57
58      p = sum([2**i * int(bit) for i, bit in enumerate(e[::-1])])
59      if p > 0:
60          p -= bias
61          is_normalized = True
62      else:
63          p = -bias + 1
64          is_normalized = False
```

```
    if p > bias:
        p = np.nan
    return p, is_normalized


def mantissa_str_to_float(
        m: str,
        is_normalized: bool,
        len_m: int = 52
    ) -> float:
    """mantissa string to mantissa integer

    Args:
        m (str): mantissa string consiting of of "0" and "1"
        is_normalized (bool): True if m is normalized number else False
        len_m (int, optional): the number of bits for the mantissa. Defaults to
            52.

    Returns:
        float: mantissa in decimal float
    """
    assert isinstance(m, str)
    assert len(m) == len_m
    for bit in m:
        assert bit in ["0", "1"]

    p = 1 if is_normalized else 0
    p += sum([2 ** -(i + 1) * int(bit) for i, bit in enumerate(m)])

    return p

def sem_to_str(s: str, e: str, m: str, n_spaces: int = 0) -> str:
    """convert output strings of `float64_to_binary()` with `separate=True` to
        a human-readable string

        Args:
        s, e, m (str): bit strings of sign, exponent, and mantissa.
        n_spaces (int, optional): the number of leading white spaces on the
            left. Defaults to 0.

    Returns:
        str: human-readable string represetnation of a number represented by s,
            e, and m
    """
    s = sign_str_to_str(s)
    e, is_normalized = exponent_str_to_int(e)
    m = mantissa_str_to_float(m, is_normalized)

    n_spaces = " " * n_spaces

    if e is not np.nan:
        return f"{n_spaces}{s} 2^{e:<+7d} * {m}"
    else:
```

```
114         return f"{n_spaces}{s} nan/inf    * {m}"
```

Listing 2　課題 02-2/a b を bit で表示する関数

そして、この関数を用い、それぞれの文字が IEEE754 binary64 でどのように表示されているか確認するコードを以下に示します。

```
print(f"decimal  s exponent    mantissa")
for f in [
        a,
        b,
        a + b,
        -b,
        c,
    ]:
    s, e, m = float64_to_binary(f, separate=True)
    print(f"{f:.2f} --> {s} {e} {m}")
    print(sem_to_str(s, e, m, n_spaces=9))
```

そして、実行結果が以下になります。

| 数 | 正負 | 指数 | 仮数 |
|---|---|---|---|
| a | 0 | 10000000011 | 0010000000000000000000000000000000000000000000000000 |
| b | 0 | 10000111000 | 0000000000000000000000000000000000000000000000000000 |
| a + b | 0 | 10000111000 | 0000000000000000000000000000000000000000000000000001 |
| -b | 1 | 10000111000 | 0000000000000000000000000000000000000000000000000000 |
| c | 0 | 10000000100 | 0000000000000000000000000000000000000000000000000000 |

## 1.4　a + b の計算

a + b の計算手順は以下です。

1. 指数の整列
2. 仮数の加算
3. 正規化
4. 丸め

この処理手順に従って説明する。

1. 指数の整列まず、a,b は表によって示されているとおりです。a の桁数は指数部分より、 $2^4$ で、b は $2^{57}$ である。仮数を計算するためにまず桁を同じにする必要があり、桁が大きい方すなわち b のほうに合わせる。桁数の合わせ方は、仮数を右に何 bit 右シフトすることで桁を合わせる。今回の場合 52bit 右シフトする必要です。右シフトをした a(表では a' とする) はこのようにます。

| 数 | 正負 | 指数 | 仮数 |
|---|---|---|---|
| a | 0 | 10000000011 | 0010000000000000000000000000000000000000000000000000 |
| a' | 0 | 10000111000 | (0.)0000000000000000000000000000000000000000000000001001 |
| b | 0 | 10000111000 | 0000000000000000000000000000000000000000000000000000 |

2. 仮数の加算桁数が同じになったため仮数を足し合わせる。また、丸みを考慮するため 52bit 以降の桁も記します。

| 数 | 正負 | 指数 | 仮数 |
|---|---|---|---|
| a’ | 0 | 10000111000 | (0.)00000000000000000000000000000000000000000000000000001001 |
| b | 0 | 10000111000 | 00000000000000000000000000000000000000000000000000000000 |
| a + b | 0 | 10000111000 | (1.)00000000000000000000000000000000000000000000000000001001 |
| (a+b)’ | 0 | 10000111000 | 00000000000000000000000000000000000000000000000000000001 |

3. 正規化正規化は、一番左の整数部分が 1 ではないときに行うものである。今回は整数部分が 1 であるため特に何も行わない。
4. 丸め丸めでは飛び出てしまった部分が 0.5 以上であればくりあげを行う。今回の場合は繰り上げられるため、結果は (a+b)’ で表示する。この時に，$b$ が非常に大きいため，$a$ の寄与が落ちてしまうのである。

# 2 課題 2-05

## 2.1 課題の内容

softamax 関数をコンピュータで実装するときに、コンピュータの計算の限界から、実際の値との誤差が生まれることを検証しました。

## 2.2 softmax 関数とは

softmax 関数とは、シグモイド関数を多次元に拡張した関数であり、多クラス分類問題において、ニューラルネットワークの出力を確率分布 (0,1) に変換することができるので、機械学習や深層学習で使用される関数である。softmax 関数 $\sigma(x)$ は次のように定義される。

$$\sigma(x) = (\sigma(x)_1, \sigma(x)_2, \ldots, \sigma(x)_M)^T$$
$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{M} e^{x_j}}$$

## 2.3 愚直に実装した softmax 関数の実装方法と結果

### 2.3.1 softmax 関数の実装方法

上式のまま python で実装したものが以下のコードである。

```python
#愚直に実装した数値的に不安定なsoftmax
def unstable_softmax(x: np.ndarray,) -> np.ndarray:
    a = np.sum(np.exp(x))
    y = np.exp(x) / a

    return y
```

Listing 3　課題 02-05/愚直に実装した softmax 関数

コードを見てみると、softmax 関数には exp をデータ数に応じて足し合わせる動作があるため、簡単にオーバーフローしてしまうことが考えられます。

### 2.3.2 結果

結果の出力には以下のコードを使用した。

```
  rng = np.random.default_rng()
a = rng.integers(800, 1000, size=10)
a = a.astype(np.float64)
print(a.dtype)

y = unstable_softmax(a)
print(y)
for i in y:
    print(f"{i:.15e}")
```

NumPy に含まれる乱数生成の default_rng でジェネレータのインスタンスを作り、インスタンスに対して、確率分布メソッドを呼び出し、型を決定して softmax 関数に入れました。実行結果は nan になり、オーバーフローしていることがわかりました。

## 2.4 演習資料で説明されていた数値的に安定な softmax 関数の実装方法と結果

### 2.4.1 演習資料で説明された softmax 関数の実装方法

愚直に実装した softmax 関数ではオーバーフローしてしまったため、softmax 関数並進不変性を利用する。

$$\begin{aligned}\sigma(x+y)_i &= \frac{e^{x_i+y}}{\sum_{j=1}^{M} e^{x_j+y}} \\ &= \frac{e^{y}e^{x_i}}{e^{y}\sum_{j=1}^{M} e^{x_j}} \\ &= \sigma(x)_i\end{aligned}$$

また、最大値 $\tilde{x} = \max_i x_i$ を用いて、log-sum-exp trick を適用します。この式を実装したものが以下のコードになります。

```
def stable_softmax(x: np.ndarray,) -> np.ndarray:
    max = np.max(x)
    exp_x = np.exp(x - max)
    sum_exp_x = np.sum(exp_x)

    y = np.exp(x - max - np.log(sum_exp_x)) #log


    return y
```

Listing 4　課題 02-05/演習資料で説明されていた数値的に安定な softmax 関数

### 2.4.2 結果

以下が結果の一例である。

1.587178613846118e-60
4.157641866918001e-07
4.999997921178430e-01
3.833822442893757e-53
1.546674362601999e-50
1.652849626184181e-37
1.271832294905762e-13

1.172775181673195e-59
4.999997921178430e-01
5.056105360903163e-44

この結果から softmax 関数がオーバーフローしないことがわかりました。また、scipy.sepecial.softmax の結果が以下のようになります。
array([1.58717861e-60, 4.15764187e-07, 4.99999792e-01, 3.83382244e-53, 1.54667436e-50, 1.65284963e-37, 1.27183229e-13, 1.17277518e-59, 4.99999792e-01, 5.05610536e-44])
より、結果が一致するため数値が安定的であるといえる。

# 3 課題 2-06

## 3.1 課題の内容

$1+\epsilon > 1$ となる（つまり演算の前後で値が変化する）ような最小の浮動小数点数 $\epsilon$ マシンイプシロンの値を求めるアルゴリズムを作成します。

## 3.2 実装方法について

以下のように関数にまとめ、どの型でも反映できるようにしました。

```
def machine_epsilon(dtype):
    eps = dtype(1)
    one = dtype(1)

    while one + eps > one:
        eps = eps / dtype(2)  # εを半分にする
        print(f"{eps:.15e}")  # 表記で表示e

    eps = eps * dtype(2)  # 最後に倍して戻す2
    print("Machine epsilon =", f"{eps:.15e}")

    return eps
```

基本的なアルゴリズムのまま作成しているため、詳細なコードの説明は省略する。

## 3.3 実行結果

アルゴリズムで計算したマシンイプシロンと np.finfo で出力した悔過を以下にしめす。np.float64
Machine epsilon = 2.220446049250313e-16
np.float32
Machine epsilon = 1.192092895507812e-07
np.float16
Machine epsilon = 9.765625000000000e-04

np.finfo()
2.220446049250313e-16
1.1920929e-07
0.000977

このように結果が一致することがわかりました。