STAT243 Lecture 8.2 Floating Point Basics

> **Logic**
>
> 关于 Floating point system 的详细论述, 见 DDA3005 Lecture 2

---

1 表示实数 (Representing Real Numbers)

1.1 初步探索 (Initial Exploration)

在计算机中，实数（即浮点数）只能被**近似表示**。虽然这种近似非常精确，但仍有细微误差。

然而，当结果非常大或非常小时，或当我们比较两个实数是否“相等”时，这种微小误差可能导致出乎意料的结果。

Python

```python
0.3 - 0.2 == 0.1      # False
0.75 - 0.5 == 0.25    # True
0.6 - 0.4 == 0.2      # True
```

为什么 `0.3 - 0.2` 与 `0.1` 不相等，而 `0.75 - 0.5` 却能成立?
这是因为 **0.1、0.2、0.3 等十进制小数无法用二进制有限位精确表示**。

1.2 精度探索 (Digits of Accuracy)

通过 `format` 或 `dg()` 函数可查看浮点数的高精度表示：

Python

```python
a = 0.3
b = 0.2
dg(a)   # 0.29999999999999998890
dg(b)   # 0.20000000000000001110
dg(a - b)   # 0.09999999999999997780
dg(0.1)     # 0.10000000000000000555
dg(1/3)     # 0.33333333333333331483
```

经验上，浮点数在 **第 16 位有效数字** 之后开始失真。
无论小数点位置在哪里，**浮点精度取决于有效位数而非小数位数**。

Python

```python
dg(1234.1234)   # 1234.12339999999994688551
dg(1234.123412341234)   # 1234.12341234123391586763
```

结果表明：
双精度浮点数（64 位）能精确表示约 **16 位有效数字**。

1.3 可精确表示的数

```python
dg(0.1)   # 不能精确表示
dg(0.5)   # 可精确表示
dg(0.25)  # 可精确表示
dg(1/32)  # 可精确表示
```

原因：只有分母为 2 的幂的分数才能在二进制下被精确表示。
例如：

- 0.5 = 1/2 = 2^{-1}
- 0.25 = 1/4 = 2^{-2}

而 0.1 在二进制下是无限循环小数，无法有限表示，就像 1/3 在十进制中一样。

---

2 机器精度 (Machine Epsilon)

Machine epsilon 表示浮点数的相对精度，定义为最小的 x 使得 $1 + x \neq 1$。

```python
1e-16 + 1.0     # 1.0
np.array(1e-16) + np.array(1.0) # np.float64(1.0)
2e-16 + 1.0     # 1.0000000000000002
dg(2e-16 + 1.0) # 1.0000000000000022204
np.finfo(np.float64).eps    # 2.220446049250313e-16
np.finfo(np.float32).eps    # 1.1920929e-07
```

解释：

- 对**双精度 (float64)**, $\varepsilon = 2^{-52} \approx 2.22 \times 10^{-16}$
- 对**单精度 (float32)**，$\varepsilon = 2^{-23} \approx 1.19 \times 10^{-7}$

这意味着浮点数在 1 附近的最小可区分间距约为 2×10^{-16}。

> ⚠ **Remark:IEEE 754 加法舍入机制**
>
> 浮点数加法的规则是：
>
> **结果会舍入到最接近的可表示浮点数（nearest representable float）。**
>
> 在 double 精度下，能表示的 1 附近的相邻两个数的间隔是：
>
> $$2^{-52} = 2.220446049250313 \times 10^{-16}$$
>
> - 如果加的数小于 **ε/2 ≈ 1.11×10⁻¹⁶**，结果会被舍入回 1。
> - 只有当加的数 ≥ ε/2，才会“跳到”下一个可表示数。
>
> 因此：
>
> ```
> 1 + 1e-16   # < ε/2 → 被舍回 1.0
> 1 + 2e-16   # > ε/2 → 跳到下一个表示数
> ```

变量	含义	float32	float64
尾数位数（不含隐藏位）	存储在内存中的小数位数	23	52
有效位数（含隐藏位）	实际精度	24	53

变量	含义	float32	float64
机器精度 ε（两种写法）		2^{-23} 或 $2^{-(24-1)}$	2^{-52} 或 $2^{-(53-1)}$

3 Floating Point Representation

3.1 浮点数的内部结构

计算机以**科学计数法的二进制形式**存储浮点数：

$$\pm d_0.d_1d_2\ldots d_p \times b^e$$

其中：

- $b = 2$（二进制基数）
- e 为指数
- d_i 为尾数位 (mantissa digits)

对于 **IEEE 754 双精度浮点数 (float64)**：

$$(-1)^S \times 1.d_1d_2\ldots d_{52} \times 2^{(e-1023)}$$

- 共 64 位 = 1 (符号位) + 11 (指数位) + 52 (尾数位)
- 隐含的前导 1 不存储（称为 *hidden bit*）

> ⚠ **Remark**
>
> 浮点数的好处在于，我们可以表示从极小到极大的数值，同时保持良好的精度。浮点数会浮动以适应数值的大小。假设我们只有三位数可用，并且使用十进制。在浮点表示法中，我们可以将 $0.12\times0.12 = 0.0144$ 表示为 $(1.20\times10-1)\times(1.20\times10-1) = 1.44\times10-2$，但如果我们将小数点固定，就会得到 $0.120\times0.120 = 0.014$，我们会损失一位精度。（此外，我们无法表示大于 0.99 的数值。）

3.2 示例：浮点数二进制结构

Python

```python
# bits 函数在上个 lecture 中被定义过
bits(2.0**(-1))  # 1/2
# '0011111111100000000000000000000000000000000000000000000000000000'
bits(2.0**0)     # 1
# '0011111111110000000000000000000000000000000000000000000000000000'
bits(2.0**1)     # 2
# '0100000000000000000000000000000000000000000000000000000000000000'
bits(2.0**2)     # 4
# '0100000000001000000000000000000000000000000000000000000000000000'
bits(5.25)
# '0100000000010101000000000000000000000000000000000000000000000000'
```

解析 5.25 的二进制表示：

$$5.25 = 1.0101_2 \times 2^2 = 1\times2^2 + 0\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2}$$

> ⚠ **Remark**
>
> 在以上代码中, 我们通过添加一个小数点 `.0` 强制存储为双精度浮点数

数值	指数位 (十进制)	真指数	尾数部分	对应数学表达式
0.5	1022	−1	0	1.0×2^{-1}
1	1023	0	0	1.0×2^{0}
2	1024	+1	0	1.0×2^{1}
4	1025	+2	0	1.0×2^{2}
5.25	1025	+2	0101...	$1.0101_2\times2^{2}$

3.3 精度与范围的权衡

给定固定位数：

- 增加尾数位 → 提高精度
- 增加指数位 → 扩大可表示范围

因此，**浮点格式是精度与范围的权衡**。

---

4 溢出与下溢 (Overflow and Underflow)

指数使用 11 位，表示范围约为 $e \in [0, 2047]$。
因此：

- 最大可表示数 ≈ $2^{1024} \approx 10^{308}$
- 最小非零数 ≈ $2^{-1022} \approx 10^{-308}$

Python

```python
import numpy as np

x = np.float64(10)

print(x**308)   # ok
# np.float64(1e+308)

x**309   # overflow
# <string>:1: RuntimeWarning: overflow encountered in scalar power np.float64(inf)

np.log10(2.0**1024) # Just barely overflows.
# OverflowError: (34, 'Numerical result out of range')

np.log10(2.0**1023)
# np.float64(307.95368556425274)

np.finfo(np.float64)
# finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308,
dtype=float64)
```

若结果超出范围，则发生：

- **Overflow**（过大 → ∞）
- **Underflow**（过小 → 0）

下溢示例：

Python

```python
x**(-308)   # ok
# np.float64(1e-308)

x**(-330)   # underflow to 0
# np.float64(0.0)
```

有趣的是，部分非常小的数（10^{-309} 到 10^{-323}）仍可表示，称为 **subnormal numbers（非正规数）**。

⚠ **Remark: Gradual underflow**

什么是非正规数（subnormal/denormal）

- 在 IEEE 754 浮点标准里，普通的（正规）浮点数满足
  $x = (-1)^s \times (1.f) \times 2^{E-bias}$
  其中尾数有隐含的首位 1（即 1.f）
- **非正规数** 出现在指数位全为 0 且尾数不为 0 的编码中，这时不再有隐含的首位 1，而是
  $x = (-1)^s \times (0.f) \times 2^{1-bias}$
  也就是把有效数字的最高位从 1 改成 0，同时把指数固定为 $1 - bias$

为什么需要非正规数：渐进下溢（gradual underflow）

- 如果没有非正规数，最小正的正规数是 2^{1-bias}，再小就直接变成 0，数轴在 0 附近会出现“断崖”
- **非正规数让最小可表示值从 2^{1-bias}** 开始，以均匀步长向 0 逐级逼近，实现所谓的 **gradual underflow**，避免突然变成 0
- 这就是 10^{-309} 到 10^{-323} 之间仍可表示的原因

双精度（float64）的具体数值

- 指数位 11 位，**bias = 1023**
- 最小正的**正规**数：$\text{tiny} = 2^{1-1023} = 2^{-1022} \approx 2.225074 \times 10^{-308}$
- 最小正的**非正规**数：$\min \text{subnormal} = 2^{1-1023} \times 2^{-52} = 2^{-1074} \approx 4.940656458 \times 10^{-324}$
- 最大的**非正规**数：$(1 - 2^{-52}) \times 2^{-1022} = 2^{-1022} - 2^{-1074}$，恰好比最小正规数小一个最小步长
- 这解释了为何 $10^{-309} \sim 10^{-323}$ 仍可表示，因为它们落在 $[2^{-1074}, 2^{-1022})$ 这个非正规区间

单精度（float32）的对应数值

- 指数位 8 位，**bias = 127**
- 最小正的**正规**数：$2^{1-127} = 2^{-126} \approx 1.175494 \times 10^{-38}$
- 最小正的**非正规**数：$2^{-126} \times 2^{-23} = 2^{-149} \approx 1.401298 \times 10^{-45}$

精度与步长：正规区 vs 非正规区

- 在**正规区**，间隔（ULP）随数值大小成比例，**相对误差**约为 machine epsilon 的量级（float64 约 2^{-52}）
- 在**非正规区**，有效位从 53 位降到更少，因为没有隐含 1，且指数固定为 $1 - bias$，**步长是常数**
  - 对于 float64，非正规区的最小步长就是 2^{-1074}，与数值本身同阶，因而**相对误差会显著变大**
- 这就是“渐进下溢”的代价：不会直接变成 0，但相对精度会变差

5 Integers or Floats

5.1 Numpy 整数

Numpy 整数类型在溢出后会**回绕 (wrap around)**：

Python

```python
np.log2(np.iinfo(np.int64).max)
# np.float64(63.0)

x = np.int64(2)

x**63+1 # 溢出
# -9223372036854775807

x**64  # 溢出
# np.int64(0)
```

Python 的 `int` 不溢出：

Python

```python
print(2**64)    # 18446744073709551616
print(2**100)   # 1267650600228229401496703205376
```

Python

```python
import sys
print(sys.getsizeof(2**10))     # 28 bytes
print(sys.getsizeof(2**1000))   # 160 bytes
print(sys.getsizeof(2**10000))  # 1448 bytes
```

5.2 用浮点数表示整数的上限

而浮点数的上限远高于整数类型：

Python

```python
x = np.float64(2)
dg(x**64, '.2f')    # 18446744073709551616.00
dg(x**100, '.2f')   # 1267650600228229401496703205376.00
```

5.3 整数在浮点中的精确存储限制

浮点数能**精确表示的整数范围**为 $|x| < 2^{53}$。

原因：

- 尾数有 52 个显式二进制位 + 1 个隐含位 → 共 53 位有效数字。
- 超出该范围时，相邻可表示整数的间距大于 1。

Python

```python
2.0**52, 2.0**53, 2.0**53+1  # 后者不再精确表示
```

5.4 强制储存为整数/双精度浮点数

可以通过几种方式强制存储为整数或双精度浮点数:

Python

```python
x = 3; type(x)
# <class 'int'>
```

Python

```python
x = np.float64(x); type(x)
# <class 'numpy.float64'>
```

Python

```python
x = 3.0; type(x)
# <class 'float'>
```

Python

```python
x = np.float64(3); type(x)
# <class 'numpy.float64'>
```

6 浮点数的精度与高精度运算 (Precision and High-Precision Numbers)

6.1 浮点数的有限精度 (Finite Precision of Floating-Point Representation)

在浮点数表示中，一个实数被存储为三部分：

$$(S, d, e)$$

其中：

- S：符号位 (Sign)
- d：尾数 (Mantissa)
- e：指数 (Exponent)

对于 **IEEE 754 双精度浮点数 (float64)**：

- 尾数部分 $p = 52$ 位
- 因此能表示 $2^{52} \approx 4.5 \times 10^{15}$ 个不同的值
- 对应约 **16 位十进制有效数字**

这意味着浮点数并非连续的，而是**离散取值**。
如果一个实数落在两个可表示数之间，它会被**舍入到最近的可表示值**。
因此浮点数的精度约为半个间隔（即半个 *gap*）。

6.1.1 示例：数值大小与有效精度

Python

```python
# 大小不同的数的精度比较
dg(.1234123412341234)
# 0.12341234123412339607

dg(1234.1234123412341234)   # not accurate to 16 decimal places
# 1234.1234123412414324131

```

```python
dg(123412341234.123412341234)   # only accurate to 4 places
# 123412341234.1234130859375000000

dg(1234123412341234.123412341234)   # no places!
# 1234123412341234.0000000000000000000

dg(12341234123412341234)    # fewer than no places!
# 12341234123412340736.000000000000000000
```

结论：

- 浮点数精度取决于**有效位数**而非小数位数。
- 当数值很大时，绝对精度下降（相邻数的间隔变大）。

6.2 数值运算中的间距效应 (Spacing and Arithmetic Implications)

当两个数非常接近时，若它们的差值小于机器可区分间距 (machine epsilon)，计算结果会被“吞没”成 0。

Python

```python
dg(1234567812345678.0 - 1234567812345677.0)
# 1.0000000000000000000
dg(12345678123456788888.0 - 12345678123456788887.0)
# 0.0000000000000000000
dg(12345678123456780000.0 - 12345678123456770000.0)
# 10240.000000000000000000
```

解释：

- 数越大，相邻可表示数之间的间距越大。
- 当差值小于间距时，浮点数无法辨别差异，结果被四舍五入。

6.3 机器精度 (Machine Epsilon)

浮点数在 1.0 附近的最小间距（即相邻可表示数的差）称为 **机器精度**，记为 ϵ。

推导如下：

$$1.000\ldots00_2\ 1.000\ldots01_2$$

二者之差：

$$\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$$

6.3.1 含义：

- ϵ 表示在数值 1 附近的**绝对间距**；
- 对任意数 x，其相对间距为：

$$\frac{(1+\epsilon)x - x}{x} = \epsilon$$

6.3.2 示例：不同数量级的绝对误差

若 $x = 10^6$，则绝对误差约为：

$$x\epsilon = 10^6 \times 2.22 \times 10^{-16} \approx 2 \times 10^{-10}$$

Python

```python
dg(1000000.1)
# 1000000.0999999999671693563
```

可以看到：
数值在 10^{-10} 量级上仍可区分，但 10^{-11} 的变化已不可检测。

6.4 整数值的精确性与浮点间距 (Exact Integers in Float64)

浮点数的精度间距为 $x\epsilon$。
对于双精度浮点数：

- 当 $x \approx 2^{52}$，间距 = 1
- 当 $x \approx 2^{53}$，间距 = 2
- 当 $x \approx 2^{54}$，间距 = 4

这意味着：

- **当整数大于 2^{53} (~9×10¹⁵) 时，浮点数无法再精确区分相邻整数。**

Python

```python
2.0**52
# 4503599627370496.0

2.0**52 + 1   # 可区分
# 4503599627370497.0

2.0**53
# 9007199254740992.0

2.0**53 + 1   # 不再区分
# 9007199254740992.0

2.0**53 + 2
# 9007199254740994.0

dg(2.0**54)
# 18014398509481984.000000000000000000

dg(2.0**54+2)
# 18014398509481984.000000000000000000

2.0**54 + 4
# 18014398509481988.000000000000000000
```

6.4.1 对应二进制表示验证：

Python

```python
bits(2**53)
# '0100001101000000000000000000000000000000000000000000000000000000'

bits(2**53 + 1)
# '0100001101000000000000000000000000000000000000000000000000000000'
```

```
bits(2**53+2)
# '0100001101000000000000000000000000000000000000000000000000000001'

bits(2**54)
# '0100001101010000000000000000000000000000000000000000000000000000'

bits(2**54+2)
# '0100001101010000000000000000000000000000000000000000000000000000'

bits(2**54 + 4)
# '0100001101010000000000000000000000000000000000000000000000000001'
```

6.5 不同数量级下的浮点间距 (Demonstration Near 0.1)

Python

```
dg(0.1234567812345678)
# 0.12345678123456779729
dg(0.12345678123456781)
# 0.12345678123456781117
dg(0.12345678123456782)
# 0.12345678123456782505
dg(0.12345678123456783)
# 0.12345678123456782505
dg(0.12345678123456784)
# 0.12345678123456783892
```

这些值在十进制下看似连续，但在二进制表示中，每一次变化都对应尾数的最低位变化：

Python

```
bits(0.1234567812345678)
# '0011111110111111100110101101110100010101110111111001101001000110'
bits(0.12345678123456781)
# '0011111110111111100110101101110100010101110111111001101001000111'
bits(0.12345678123456782)
# '0011111110111111100110101101110100010101110111111001101001001000'
```

这体现了浮点数在底层存储中**离散分布的本质**。

7 高精度浮点数 (Working with Higher-Precision Numbers)

Python 原生的 `int` 支持任意精度，不会溢出。
但标准浮点数 (float64) 限于 16 位十进制有效数字。
若需要更高精度浮点运算，可使用 **gmpy2** 包。

7.1 示例：

Python

```
import gmpy2
gmpy2.get_context().precision = 200  # 设置200位精度
gmpy2.const_pi()                     # 高精度π
gmpy2.mpfr(".1234567812345678")      # 以MPFR类型存储
```

⚠ Remark

说明：

- `mpfr` 表示“多精度浮点数”。
- R 语言中整数默认为 4 字节，不具备此特性；
  若需要精确结果，R 通常使用 `numeric` 类型（即双精度）。