

| STAT243 Lecture 8 Precision (Reduced)

| 1 位与字节 (Bits and Bytes)

- **Bit (位)**: 最小存储单位, 可取 0 或 1 (相当于一个开关)。
- **Byte (字节)**: 由 8 位组成。

| 1.1 ASCII 文本编码

- ASCII 使用 1 个字节 (8 位) 存储一个字符, 但实际仅用到 7 位, 可表示 **128 (2^7)** 个字符。
- 字节的值范围是 0–255 (共 256 个值)。
- 简洁起见, 我们通常用 **十六进制 (hexadecimal)** 表示字节内容, 如: `3e`, `a0`, `ba` 等。
- **文件格式 (file format)** 只是对文件中字节序列的一种解释方式。

| 2 查看二进制表示 (Binary Representation with `bitstring`)

示例函数:

```
Python
1  from bitstring import Bits
2
3  def bits(x, type='float', len=64):
4      if type == 'float':
5          obj = Bits(float=x, length=len)
6      elif type == 'int':
7          obj = Bits(int=x, length=len)
8      else:
9          return None
10     return obj.bin
```

| 2.1 示例: 字符的 ASCII 编码

```
Python
1  Bits(bytes=b'a').bin # '01100001'
2  Bits(bytes=b'b').bin # '01100010'
3  Bits(bytes=b'0').bin # '00110000'
4  Bits(bytes=b'1').bin # '00110001'
5  Bits(bytes=b'2').bin # '00110010'
6  Bits(bytes=b'@').bin # '01000000'
```

说明:

- `'b'` 的编码比 `'a'` 大 1
- `'1'` 的编码比 `'0'` 大 1

3 整数的二进制存储 (Integer Representation)

- 两字节 (16 bits) → 可表示 0 到 $2^{16} - 1 = 65535$ (无符号整数)。
- 若使用一个 bit 作为符号位 (正负号), 则可表示约 -32768 到 32767 。

实际中, 负数并非以“符号位 + 数字”的形式存储, 而是采用 **二进制补码 (two's complement)** 表示, 便于计算机执行加减法。

3.1 示例: 64 位整数表示

```
Python
1 import numpy as np
2 np.binary_repr(0, width=64)
3 # '0000000000000000000000000000000000000000000000000000000000000000'
4
5 np.binary_repr(1, width=64)
6 # '0000000000000000000000000000000000000000000000000000000000000001'
7
8 np.binary_repr(2, width=64)
9 # '0000000000000000000000000000000000000000000000000000000000000010'
10
11 np.binary_repr(-1, width=64)
12 # '1111111111111111111111111111111111111111111111111111111111111111'
```

说明:

在补码表示中, -1 的二进制为全 1, 便于计算 (例如 $1 + (-1) = 0$)。

4 整数溢出 (Integer Overflow)

计算机中的整数集并非对加法封闭。若结果超出可表示范围, 将发生 **溢出 (overflow)**。

```
Python
1 a = np.int32(3423333)
2 a * a # overflow warning
3 # np.int32(-1756921895)
4
5 a = np.int64(3423333)
6 a * a
7 # np.int64(11719208828889)
```

- `int32` 溢出, 而 `int64` 不会 (因为范围更大)。

```
Python
1 a = np.int64(34233332342343)
2 a * a
3 # np.int64(1001093889201452977) # 溢出结果错误
4
5 a = 34233332342343
6 a * a
7 # 1171921043261307270950729649 # 正确
```

Python 的原生 `int` 类型支持任意精度整数, 不会溢出, 但代价是使用更多内存。

例如:



Python

```
1 import sys
2 a = 34233332342343
3 a * a
4 # 1171921043261307270950729649
5
6 sys.getsizeof(a)      # 32 bytes
7 sys.getsizeof(a * a)  # 36 bytes
```

5 浮点数 (Floating Point Numbers)

- 在 C / NumPy 中常见的实数类型：
 - float32**: 单精度 (4 字节)
 - float64**: 双精度 (8 字节)
- GPU 运算中常偏向使用单精度以节省内存与提升速度。

5.1 示例：内存占用



Python

```
1 import numpy as np, sys
2 x = np.random.normal(size=100000)
3 sys.getsizeof(x)  # float64, ~800 KB
4
5 x = np.array(np.random.normal(size=100000), dtype="float32")
6 sys.getsizeof(x)  # ~400 KB
7
8 x = np.array(np.random.normal(size=100000), dtype="float16")
9 sys.getsizeof(x)  # ~200 KB
```

规律：每降低一倍字节长度，内存占用近似减半。

5.2 内存估算公式

若使用双精度浮点数：

$$\text{Memory (MB)} = \frac{N \times 8}{10^6}$$

其中 N 为元素个数。

注意：有时计算机使用 **MiB (Mebibyte)**，即 $1, \text{MiB} = 2^{20} \text{ bytes}$ 。

6 NumPy 数值信息查询 (Machine Information)

6.1 使用 `iiinfo` 函数查询数值范围

NumPy 提供辅助函数查看不同数据类型的数值范围：



Python

```

1 np.iinfo(np.int32)
2 # iinfo(min=-2147483648, max=2147483647, dtype=int32)
3
4 np.iinfo(np.int64)
5 # iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)

```

以 32-bit (4-byte) integer 为例, **最大整数为 $2147483647 = 2^{31} - 1$** , 如果考虑正数负数和零的话, 总共可以表示 $2 \cdot 2^{31} = 2^{32}$ 个数

6.2 示例: 32 位整数边界

```

Python
1 np.binary_repr(2147483647, width=32)
2 # '01111111111111111111111111111111' # 最大正数
3
4 np.binary_repr(-2147483648, width=32)
5 # '10000000000000000000000000000000' # 最小负数

```

尝试超过范围会报错:

```

Python
1 np.int32(2147483648)
2 # OverflowError: Python integer 2147483648 out of bounds for int32

```

7 表示实数 (Representing Real Numbers)

7.1 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
1 0.3 - 0.2 == 0.1 # False
2 0.75 - 0.5 == 0.25 # True
3 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**。

7.2 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
1 a = 0.3
2 b = 0.2
3 dg(a) # 0.2999999999999998890
4 dg(b) # 0.2000000000000001110
5 dg(a - b) # 0.0999999999999997780
6 dg(0.1) # 0.1000000000000000555
7 dg(1/3) # 0.3333333333333331483

```

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
1 dg(1234.1234) # 1234.12339999999994688551
2 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**。

7.3 可精确表示的数

```
Python
1 dg(0.1) # 不能精确表示
2 dg(0.5) # 可精确表示
3 dg(0.25) # 可精确表示
4 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。

8 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
1 1e-16 + 1.0 # 1.0
2 np.array(1e-16) + np.array(1.0) # np.float64(1.0)
3 2e-16 + 1.0 # 1.0000000000000002
4 dg(2e-16 + 1.0) # 1.00000000000000022204
5 np.finfo(np.float64).eps # 2.220446049250313e-16
6 np.finfo(np.float32).eps # 1.1920929e-07
```

解释：

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

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：

$$2^{-52} = 2.220446049250313 \times 10^{-16}$$

解析 5.25 的二进制表示:

 Remark

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...	$1.0101_2 \times 2^2$

9.3 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。

10 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 $\approx 2^{1024} \approx 10^{308}$
- 最小非零数 $\approx 2^{-1022} \approx 10^{-308}$



Python

```
1 import numpy as np
2
3 x = np.float64(10)
4
5 print(x**308)    # ok
6 # np.float64(1e+308)
7
8 x**309    # overflow
9 # <string>:1: RuntimeWarning: overflow encountered in scalar power np.float64(inf)
10
11 np.log10(2.0**1024) # Just barely overflows.
```

```

12 # OverflowError: (34, 'Numerical result out of range')
13
14 np.log10(2.0**1023)
15 # np.float64(307.95368556425274)
16
17 np.finfo(np.float64)
18 # 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 $\rightarrow \infty$)
- **Underflow** (过小 $\rightarrow 0$)

下溢示例：

```

Python
1 x**(-308) # ok
2 # np.float64(1e-308)
3
4 x**(-330) # underflow to 0
5 # 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**：tiny = $2^{1-1023} = 2^{-1022} \approx 2.225074 \times 10^{-308}$
- 最小正的**非正规数**：min 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

11 如何表示整数: Integers or Floats?

11.1 Numpy 整数

Numpy 整数类型在溢出后会**回绕 (wrap around)**:

```
Python
1 np.log2(np.iinfo(np.int64).max)
2 # np.float64(63.0)
3
4 x = np.int64(2)
5
6 x**63 # 溢出
7 # -9223372036854775808
8
9 x**64 # 溢出
10 # np.int64(0)
```

Python 的 `int` 不溢出:

```
Python
1 print(2**64) # 18446744073709551616
2 print(2**100) # 1267650600228229401496703205376
```

```
Python
1 import sys
2 print(sys.getsizeof(2**10)) # 28 bytes
3 print(sys.getsizeof(2**1000)) # 160 bytes
4 print(sys.getsizeof(2**10000)) # 1448 bytes
```

11.2 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
1 x = np.float64(2)
2 dg(x**64, '.2f') # 18446744073709551616.00
3 dg(x**100, '.2f') # 1267650600228229401496703205376.00
```

11.3 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
1  2.0**52, 2.0**53, 2.0**53+1  # 后者不再精确表示
```

11.4 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
1  x = 3; type(x)
2  # <class 'int'>
```

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

```
Python
1  x = 3.0; type(x)
2  # <class 'float'>
```

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

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

12.1 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*）。

12.1.1 示例：数值大小与有效精度

```
Python
1 # 大小不同的数的精度比较
2 dg(.1234123412341234)
3 # 0.12341234123412339607
4
5 dg(1234.1234123412341234) # not accurate to 16 decimal places
6 # 1234.12341234123414324131
7
8 dg(123412341234.123412341234) # only accurate to 4 places
9 # 123412341234.12341308593750000000
10
11 dg(1234123412341234.123412341234) # no places!
12 # 1234123412341234.00000000000000000000
13
14 dg(12341234123412341234) # fewer than no places!
15 # 12341234123412340736.00000000000000000000
```

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）。

12.2 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
1 dg(1234567812345678.0 - 1234567812345677.0)
2 # 1.000000000000000000000000
3 dg(1234567812345678888.0 - 1234567812345678887.0)
4 # 0.000000000000000000000000
5 dg(12345678123456780000.0 - 12345678123456770000.0)
6 # 10240.0000000000000000000000
```

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。

12.3 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：

$$1.000\dots00_2 \quad 1.000\dots01_2$$

二者之差：

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

12.3.1 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：

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

12.3.2 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

```
1 dg(1000000.1)
2 # 1000000.09999999997671693563
```

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。

12.4 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} ($\sim 9 \times 10^{15}$) 时，浮点数无法再精确区分相邻整数。

 Python

```
1 2.0**52
2 # 4503599627370496.0
3
4 2.0**52 + 1 # 可区分
5 # 4503599627370497.0
6
7 2.0**53
8 # 9007199254740992.0
9
10 2.0**53 + 1 # 不再区分
11 # 9007199254740992.0
12
13 2.0**53 + 2
14 # 9007199254740994.0
15
16 dg(2.0**54)
17 # 18014398509481984.00000000000000000000
18
19 dg(2.0**54+2)
```

```
20 # 18014398509481984.00000000000000000000
21
22 2.0**54 + 4
23 # 18014398509481988.00000000000000000000
```

12.4.1 对应二进制表示验证:

```
Python
1 bits(2**53)
2 # '0100001101000000000000000000000000000000000000000000000000000000'
3
4 bits(2**53 + 1)
5 # '0100001101000000000000000000000000000000000000000000000000000001'
6
7 bits(2**53+2)
8 # '0100001101000000000000000000000000000000000000000000000000000011'
9
10 bits(2**54)
11 # '0100001101010000000000000000000000000000000000000000000000000000'
12
13 bits(2**54+2)
14 # '0100001101010000000000000000000000000000000000000000000000000011'
15
16 bits(2**54 + 4)
17 # '0100001101010000000000000000000000000000000000000000000000000111'
```

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

```
Python
1 dg(0.1234567812345678)
2 # 0.12345678123456779729
3 dg(0.12345678123456781)
4 # 0.12345678123456781117
5 dg(0.12345678123456782)
6 # 0.12345678123456782505
7 dg(0.12345678123456783)
8 # 0.12345678123456782505
9 dg(0.12345678123456784)
10 # 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
1 bits(0.1234567812345678)
2 # '001111110111111100110101110100010101110111110011010010000110'
3 bits(0.12345678123456781)
4 # '001111110111111100110101110100010101110111110011010010000111'
5 bits(0.12345678123456782)
6 # '001111110111111100110101110100010101110111110011010010001000'
```

这体现了浮点数在底层存储中**离散分布**的本质。

| 13 计算机算术 ≠ 数学算术 (Computer Arithmetic Is Not Mathematical Arithmetic)

在数学中，实数的加、减、乘、除是**封闭的运算**，但计算机中的算术却不是。
由于数值在内存中只能被近似表示，计算机运算可能：

- 溢出 (overflow)：结果超出可表示范围
- 下溢 (underflow)：结果太小而被当作 0
- 违反代数性质，例如结合律与分配律

例如：

```
Python
1  val1 = 1/10; val2 = 0.31; val3 = 0.57
2  res1 = val1 * val2 * val3
3  res2 = val3 * val2 * val1
4  res1 == res2
5  # False
6
7  dg(res1)
8  # 0.01766999999999999821
9  dg(res2)
10 # 0.01767000000000000168
```

结果可能不同，因为浮点数运算存在微小的舍入误差，导致 $(a + b) + c \neq a + (b + c)$ 或 $a(b + c) \neq ab + ac$ 。

| 14 整数与浮点运算的差异 (Integer vs Floating-Point Calculations)

- **整数计算：**
精确且快速（但可能溢出）。
在 Python 的 `int` 类型中不会溢出，但在其他语言或 `numpy` 的定长整数中可能出现溢出。
- **浮点数计算：**
近似且较慢。
每个浮点运算 (addition/multiplication) 称为一次 **flop (floating point operation)**。
计算强度常用 flops 数量衡量。

| 15 浮点比较问题 (Comparisons)

在计算机中，**不要轻易使用 == 比较浮点数**，除非满足以下条件：

1. `x` 和 `y` 都是整数；
2. `x` 和 `y` 是整数值，但以 `double` 存储且足够小以精确表示；
3. 两个数是通过**相同的计算方式**得到的，例如：

```
Python
1  0.4 - 0.3 == 0.4 - 0.3 # True
2  0.4 - 0.3 == 0.1       # False
```

| 15.1 使用机器精度判断“近似相等”

如果两个数之间的相对差距小于机器精度 ϵ ，则认为它们“近似相等”。

通常可使用比 ε 略大的常数（例如 10ε ）以增强稳健性。

数据中常用特殊整数（如 **-9999**）代表缺失值：

这种比较是安全的，因为整数 `-9999` 可被浮点数精确表示。
但更稳妥的做法是：在读入时使用整数或字符串类型判断，再转为浮点数。

当两个接近的数相减时，大部分有效位会被抵消，剩余的位数全是舍入误差。

误差估计:

因此结果仅保留约 5 位有效数字。

实际结果略有偏差，仅保留约 28 位有效数字（而非理想的 36 位）。

16.2 计算方差的示例 (Variance Computation)

计算方差的两种写法：

```
Python
1 x = np.array([-1.0, 0.0, 1.0])
2 n = len(x)
3 np.sum(x**2) - n*np.mean(x)**2          # 方法一
4 # np.float64(2.0)
5 np.sum((x - np.mean(x))**2)            # 方法二
6 # np.float64(2.0)
```

当数据加上一个大常数：

```
Python
1 x = x + 1e8
2 np.sum(x**2) - n*np.mean(x)**2          # 错误！
3 # np.float64(0.0)
4 np.sum((x - np.mean(x))**2)            # 正确
5 # np.float64(2.0)
```

结论：

- 第一种写法会发生灾难性抵消（因 \bar{x}^2 与 $\sum x_i^2$ 数值巨大且相近）
- 第二种写法稳定得多

经验法则：

- 在运算中**先减去均值**或相似数量级的值，能显著减少误差。

16.3 数量级差异太大的加减法

当两个数差距过大时，小数部分的贡献可能被完全舍入掉。

```
Python
1 dg(123456781234.2)
2 # 123456781234.19999694824218750000
3
4 dg(123456781234.2 - 0.1)      # truth: 123456781234.1
5 # 123456781234.09999084472656250000
6
7 dg(123456781234.2 - 0.000001) # truth: 123456781234.199999
8 # 123456781234.19999694824218750000
9
10 123456781234.2 - 0.000001 == 123456781234.2
11 # True
```

此时较小的数已被“吞没”，因为：

$$x = 10^{11}, \quad \epsilon x \approx 10^{-5}$$

任何小于 10^{-5} 的变化都不会影响结果。

改进策略：

- 若要对大量不同数量级的数求和，**先排序再相加（从小到大）**

- 或采用**树状求和 (pairwise summation)**，使每次加法在相近数量级间进行。

17 对数运算的稳定性 (Log-Scale Computations)

当涉及大量数的连乘或连除时，直接计算容易溢出或下溢。

解决方案：**在对数尺度上进行计算**。

例如：

$$\frac{\prod_i x_i}{\prod_j y_j} = \exp \left(\sum_i \log x_i - \sum_j \log y_j \right)$$

多数情况下我们甚至无需取指数（如最大化对数似然时）。

17.1 log-sum-exp 技巧

问题背景

我们经常需要计算如下形式的表达式：

$$\log \left(\sum_{i=1}^n e^{x_i} \right)$$

问题在于：

- 如果某个 x_i 很大（如 $x_i = 1000$ ），那么 e^{x_i} 会导致溢出 (overflow)；
- 如果所有 x_i 都很小（如 $x_i = -1000$ ），那么 e^{x_i} 会接近 0，导致下溢 (underflow)。

基本思路

我们利用一个简单的代数恒等式来“拉回尺度”：

$$\log \left(\sum_i e^{x_i} \right) = c + \log \left(\sum_i e^{x_i - c} \right)$$

其中


$$c = \max_i x_i$$

常见应用场景

1. Softmax 函数

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

实现时通常写作：

```
 Python
1 def softmax(x):
2     c = np.max(x)
3     exp_x = np.exp(x - c)
4     return exp_x / np.sum(exp_x)
```

2. 数值积分 / 预测密度

在计算

$$\log \frac{1}{m} \sum_{j=1}^m e^{v_j}$$

时使用：

$$\log \frac{1}{m} \sum_{j=1}^m e^{v_j} = -\log m + c + \log \sum_j e^{v_j - c}$$

17.2 实例 1：多分类逻辑回归 (Multiclass Logistic Regression)

$$p_j = \frac{\exp(z_j)}{\sum_{k=1}^K \exp(z_k)}$$

若 z_k 很大或很小，会导致数值溢出。

可通过以下变形稳定计算：

$$p_j = \frac{\exp(z_j - c)}{\sum_k \exp(z_k - c)}, \quad c = \max_k z_k$$

该技巧即为 **log-sum-exp trick** 的核心思想。

17.3 实例 2：预测密度计算 (Predictive Density in Bayesian Models)

$$\begin{aligned} f(y^*|y, x) &= \int f(y^*|y, x, \theta) \pi(\theta|y, x) d\theta \\ &\approx \frac{1}{m} \sum_{j=1}^m \prod_{i=1}^n f(y_i^*|x, \theta_j) \\ &= \frac{1}{m} \sum_{j=1}^m \exp \sum_{i=1}^n \log f(y_i^*|x, \theta_j) \\ &\equiv \frac{1}{m} \sum_{j=1}^m \exp(v_j) \end{aligned}$$

若 v_j 很小（如 -1000 ），直接取 $\exp(v_j)$ 将导致下溢。

正确做法：

$$\log f(y^*|y, x) \approx \log \frac{1}{m} \sum_j \exp(v_j) = c + \log \frac{1}{m} \sum_j \exp(v_j - c)$$

其中 $c = \max_j v_j$ 。

这同样利用了 log-sum-exp 技巧，避免了浮点下溢。