Lecture 2 — Numbers in memory

C1 Research Computing — MPhil in Data Intensive Science, Michaelmas 2026

Contents

1 Bits, bytes and integers **1**
- 1.1 Bits, bytes and words . . . 1
- 1.2 Integers of fixed width . . . 2
- 1.3 Memory sizes and units . . . 3

2 Real numbers **4**
- 2.1 The IEEE 754 formats . . . 4
- 2.2 Decimal fractions in binary . . . 6
- 2.3 Narrower formats for machine learning . . . 7

3 Consequences of finite precision **7**

Troubleshooting **8**

Key points **8**

Exercises **10**

Solutions to the exercises **11**

A number in a computer is a pattern of bits of fixed width, stored in a finite memory. The width of the pattern sets the range of the integers that can be stored and the precision of the real numbers. A counter that exceeds its range turns negative without any error message. A sum of `float32` values is wrong from the seventh digit. An array stored as `float64` takes twice the memory of the same data as `float32`.

The running example is the file `C9.csv` of the `attractor` project: 141 samples of a predator–prey experiment, with the time in days, the algae and rotifer densities and four further columns [1]. Its first algae value, 1.92, is followed through the chapter from its decimal spelling to its 64 bits. The script `bits_and_floats.py` in the project folder prints every result of Sections 1.1 to 3 in one run. The outputs shown were produced with Python 3.12 and NumPy 2.5. Where a printed form depends on the version, the text says so.

1 Bits, bytes and integers

1.1 Bits, bytes and words

Definition 1.1: Bit, byte and word

A **bit** is a 0 or a 1 [2]. A **byte** is a group of 8 bits, hence $2^8 = 256$ distinct patterns. A **word** is the group of bits that a processor treats as its natural unit of data: the width of its registers, of a memory address, and of the quantity it loads, stores or adds in one instruction [3].

Current processors have 64-bit words, that is 8 bytes. The desktop processors of the 1990s and early 2000s had 32-bit words. The terms “64-bit” and “32-bit” for machines and programs refer to this width.

A pattern of n bits $b_{n-1} \dots b_1 b_0$ read as an unsigned integer has the value

$$N = \sum_{i=0}^{n-1} b_i \, 2^i, \tag{1}$$

which ranges from 0 to $2^n - 1$. The number of samples in `C9.csv`, 141, is $128 + 8 + 4 + 1 = 10001101_2$. Listing 1 shows the conversions in Python: `bin` writes an integer in base 2, `int` with a second argument reads one, and a literal with the prefix `0b` is written directly in binary [4].

> b_0 in Equation (1) is bit 0, the least significant bit, with weight $2^0 = 1$. It has nothing to do with the bias of the floating-point formats in Section 2.1, a constant added to the exponent.

```
>>> bin(141), int("10001101", 2), 0b10001101
('0b10001101', 141, 141)
>>> (141).bit_length(), (141).to_bytes(2, "big")
(8, b'\x00\x8d')
```

Listing 1: An integer in binary: 141 needs 8 bits, written as the two bytes `00 8d` in hexadecimal.

Three widths come up constantly, because the hardware is built around them: the byte, 32 bits and 64 bits.

- The byte is the smallest addressable unit. Every location in memory holds one byte and has an **address**, a number counting the bytes from 0 upwards. Sizes of files and of memory are counted in bytes, and a character in ASCII is one byte.
- **Registers** are the few dozen storage slots inside the processor. An instruction operates only on registers, so $a + b$ is: load a from memory into a register, load b into another, add, store the result back. On a 64-bit processor (arm64 on a Mac, x86-64 on a Linux cluster) a register holds 64 bits: one instruction handles an integer up to $2^{64} - 1$ or one `float64`.
- Addresses are also 64 bits wide today, so a program can name 2^{64} bytes of memory. A 32-bit address, the norm until the mid-2000s, takes $2^{32} = 4\,294\,967\,296$ values. Each address names one byte, so a program could address 2^{32} bytes $= 4$ GiB. The limit applied to the **address space** of one program rather than to the physical memory of the machine. Some 32-bit systems reached more memory through extensions such as Intel’s PAE [5], while each program still saw at most 4 GiB. With 64-bit addresses the limit is 2^{64} bytes $= 16$ EiB, beyond any machine.

Listing 2 makes addresses visible. A NumPy array is one contiguous block of bytes, and the address of element (i, j) of a 141×7 `float64` array is base $+ (7i + j) \times 8$. The first byte of the array sits at address `0x14d26c600`, a 64-bit number written in hexadecimal. The value 1.92, element $(0, 1)$, starts 8 bytes later. The next row starts 56 bytes further on, as the `strides` say. The addresses differ at every run, because the operating system places the program’s memory where it finds room.

1.2 Integers of fixed width

A Python `int` has no fixed width: `2**100` is an ordinary value, stored in as many bytes as it needs. A NumPy integer has a fixed number of bytes, chosen by its **dtype** [6]. The types are named by their width and signedness. `uint8` is an unsigned 8-bit integer, from 0 to 255,

```
>>> import numpy as np
>>> d = np.genfromtxt("C9.csv", delimiter=",", skip_header=1)
>>> hex(d.ctypes.data), d.itemsize, d.strides
('0x14d26c600', 8, (56, 8))
>>> hex(d[0, 1:2].ctypes.data), d[0, 1]
('0x14d26c608', np.float64(1.92))
```

Listing 2: Addresses of the elements of the `C9.csv` array. `ctypes.data` is the address of the first byte, `itemsize` the bytes per element, `strides` the bytes between consecutive rows and columns.

and `int8` a signed one. The wider types `int16`, `int32` and `int64`, the default, have `uint` counterparts. A signed type reserves one bit for the sign, in the **two's complement** convention [7], so `int8` ranges from -128 to 127.

Definition 1.2: Fixed-width integer and wrap-around

A fixed-width integer type of n bits stores a value by its lowest n bits. A value outside the type's range is not rejected but **wrapped around**: the pattern is kept and read back in the range of the type, so that arithmetic is done modulo 2^n. An `int8` holds -128 to 127, a `uint8` 0 to 255, an `int32` -2^{31} to $2^{31}-1$.

The range of a type depends on its sign. 141 fits in a `uint8`, 0 to 255, and wraps in an `int8`, -128 to 127. In an unsigned type no bit is spent on the sign, which is fixed to plus.

Listing 3 stores 141 as an `int8`: 10001101_2 does not fit in the 7 value bits, and the pattern reads as $141 - 256 = -115$. The same happens to a counter: an `int32` holds at most $2^{31}-1 = 2\,147\,483\,647$, and adding 1 to that value gives $-2\,147\,483\,648$. NumPy 2 emits a `RuntimeWarning` for the overflow of a scalar operation and stays silent for an array, so the wrong value is easy to miss.

```
>>> np.array([141]).astype(np.int8)[0]
np.int8(-115)
>>> np.iinfo(np.int8).min, np.iinfo(np.int8).max, np.iinfo(np.uint8).max
(-128, 127, 255)
>>> np.array([2147483647], dtype=np.int32) + 1
array([-2147483648], dtype=int32)
```

Listing 3: Wrap-around of fixed-width integers. The range of a type is read from `np.iinfo`.

1.3 Memory sizes and units

Definition 1.3: Decimal and binary prefixes

Sizes in bytes are written with one of two families of prefixes. The **decimal prefixes** of the SI system are powers of 1000: kB, MB, GB and TB are 10^3, 10^6, 10^9 and 10^{12} bytes. The **binary prefixes** defined by the IEC [8] are powers of 1024: KiB, MiB, GiB and TiB are 2^{10}, 2^{20}, 2^{30} and 2^{40} bytes, pronounced kibibyte, mebibyte, gibibyte and tebibyte. A lower-case **b** is a bit, an upper-case **B** a byte.

The two families drift apart by 2.4% per step, because $1024/1000 = 1.024$ (Table 1). Disk manufacturers, network speeds and file sizes in `ls -l` use decimal units. Memory sizes, `du -h` and most operating-system displays use binary units, often with decimal-looking symbols such as "K" and "G".

Network and internet speeds are quoted in bits per second, Mb/s and Gb/s. Memory and disk bandwidths are quoted in bytes per second, GB/s. A 100 Mb/s link moves $100/8 = 12.5$ MB per second, and a "1 Gb/s" connection downloads a 1 GB file in about 8 s, not 1 s.

Table 1: Decimal and binary prefixes for bytes. The last column is the ratio of the binary unit to the decimal unit of the same name.

decimal		bytes	binary		bytes	ratio
kilobyte	kB	10^3	kibibyte	KiB	$2^{10} = 1\,024$	1.024
megabyte	MB	10^6	mebibyte	MiB	$2^{20} = 1\,048\,576$	1.049
gigabyte	GB	10^9	gibibyte	GiB	$2^{30} = 1\,073\,741\,824$	1.074
terabyte	TB	10^{12}	tebibyte	TiB	$2^{40} = 1\,099\,511\,627\,776$	1.100

The size of an array is the number of elements times the bytes per element, available as `nbytes` [9]. The 141×7 `float64` values of `C9.csv` take $141 \times 7 \times 8 = 7\,896$ bytes (Listing 4). A $10^4 \times 10^4$ `float64` matrix takes 800 MB, and the same matrix in `float32` 400 MB. Two memories are quoted for a computer: the **RAM**, the working memory erased at power-off, and the **disk**, the persistent storage. The laptop used for these notes, a MacBook with an M4 Max processor, has $137\,438\,953\,472 = 128 \times 2^{30}$ bytes, that is 128 GiB, of RAM and a 1 TB disk. On macOS, `sysctl -n hw.memsize` prints the RAM size in bytes and `df -h` the disk size. How data moves between disk, RAM and the processor is the subject of the next chapter.

```
$ ls -l C9.csv
-rw-r--r--  1 boris  staff  5566  8 Oct 22:25 C9.csv
$ du -h C9.csv
8.0K    C9.csv
$ python -c "
import numpy as np
d = np.genfromtxt('C9.csv', delimiter=',', skip_header=1)
print(d.shape, d.dtype, d.nbytes, d.astype(np.float32).nbytes)"
(141, 7) float64 7896 3948
```

Listing 4: Three sizes of the same data. `ls -l` prints the length of the file in bytes, `du -h` the disk space it occupies, rounded up to whole 4 KiB blocks; the array holds every value as a `float64`.

The text file is smaller than the array because most of its values have only three significant digits. A number in a text file costs one byte per character. In a `float64` array it costs 8 bytes whatever its digits.

`ls -lh` prints the 5566-byte file as `5.4K`, the length divided by 1024, and `du -h` prints `8.0K` because it counts disk blocks of 4 KiB. Division by 8 converts bits to bytes and plays no role in file sizes. The date printed by `ls -l` is the time of last modification, not the creation time; `stat` prints both.

2 Real numbers

2.1 The IEEE 754 formats

Definition 2.1: Floating-point number

A real number is stored in scientific notation in base 2,

$$x = (-1)^s \times 1.f \times 2^e, \tag{2}$$

with a **sign bit** s, a **fraction** f, the bits after the leading 1, which is not stored, and an **expo-**

nent e. The exponent is stored shifted by a constant **bias**, so that the field is a non-negative integer. The shift lets a float be compared and sorted by its bit pattern, as explained below. The widths of the three fields define the format.

The IEEE 754 standard [10], first published in 1985, fixes the widths of the fields for the formats in use (Figure 1). Every CPU and GPU implements it.

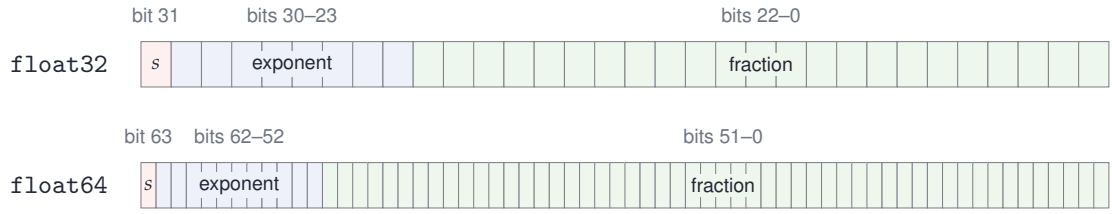


Figure 1: Bit layout of the two IEEE 754 formats: the sign bit, the exponent field, 8 or 11 bits, and the fraction, 23 or 52 bits. The exponent field holds $e + \text{bias}$.

Table 2: The IEEE 754 formats and the narrower formats of machine learning (Section 2.3). The precision is the relative spacing of adjacent values, $2^{-\text{fraction bits}}$.

format	bits	exponent	fraction	bias	precision	largest value
`float64`, fp64	64	11	52	1023	$2^{-52} \approx 2.2 \times 10^{-16}$, 16 digits	1.8×10^{308}
`float32`, fp32	32	8	23	127	$2^{-23} \approx 1.2 \times 10^{-7}$, 7 digits	3.4×10^{38}
bf16	16	8	7	127	$2^{-7} \approx 7.8 \times 10^{-3}$, 3 digits	3.4×10^{38}
`float16`, fp16	16	5	10	15	$2^{-10} \approx 9.8 \times 10^{-4}$, 3 digits	65 504

The exponent e of Equation (2) is the power of 2 that brings the number into the interval $[1, 2)$. It is positive for numbers of 2 or more and negative for numbers below 1: $8.0 = 1.0 \times 2^3$, $1.92 = 1.92 \times 2^0$, $0.5 = 1.0 \times 2^{-1}$ and $0.1 \approx 1.6 \times 2^{-4}$. A field of bits holds a non-negative integer, so a negative exponent needs a convention. IEEE 754 stores the sum $e + \text{bias}$, with the bias equal to $2^{k-1} - 1$ for a k-bit field: $2^7 - 1 = 127$ in `float32` and $2^{10} - 1 = 1023$ in `float64`. The stored field is then an unsigned integer with $e = 0$ at its middle value. Table 3 works through the `float32` field for a range of values, and Listing 5 reads the same fields from the bit patterns.

The bias of `float64` is 1023, not 1024. The formula is $2^{k-1} - 1$ for a k-bit field: 127 for the 8 bits of `float32` and 1023 for the 11 bits of `float64`.

Table 3: The `float32` exponent field, $e + 127$, for nine values. The last column is the whole 32-bit pattern read as an unsigned integer: it increases with the value.

value	normalised form	e	field $e + 127$	pattern as an integer
1.2×10^{-38}	1.02×2^{-126}	-126	$1 = 00000001_2$	8 563 486
0.1	1.6×2^{-4}	-4	$123 = 01111011_2$	1 036 831 949
0.5	1.0×2^{-1}	-1	$126 = 01111110_2$	1 056 964 608
1.0	1.0×2^{0}	0	$127 = 01111111_2$	1 065 353 216
1.92	1.92×2^{0}	0	$127 = 01111111_2$	1 073 070 735
2.0	1.0×2^{1}	1	$128 = 10000000_2$	1 073 741 824
8.0	1.0×2^{3}	3	$130 = 10000010_2$	1 090 519 040
1000.0	1.95×2^{9}	9	$136 = 10001000_2$	1 148 846 080
3.4×10^{38}	2.00×2^{127}	127	$254 = 11111110_2$	2 139 081 118

The convention has two consequences visible in the last column of Table 3. The 32-bit patterns of positive floats, read as unsigned integers, are in the same order as the values: the exponent

```
>>> def field(x):
...     u = struct.unpack(">I", struct.pack(">f", x))[0]
...     return (u >> 23) & 0xFF, u
>>> [field(x) for x in (0.1, 1.0, 1.92, 8.0)]
[(123, 1036831949), (127, 1065353216), (127, 1073070735), (130, 1090519040)]
>>> field(8.0)[0] - 127
3
```

Listing 5: The exponent field of a `float32`, read from its 32 bits: the 8 bits after the sign bit, as an integer, minus 127.

field comes first, a larger exponent means a larger number, and at equal exponent a larger fraction does. A processor or a sorting routine can therefore compare two positive floats with one integer comparison, and only the sign bit needs separate treatment. The all-zero pattern is $+0.0$, so a block of memory cleared to zero is an array of zeros.

The field values 0 and 255 are reserved. Field 0 encodes zero and the subnormal numbers, the values below 2^{-126} that are stored with a leading 0 instead of the implicit leading 1. Field 255 encodes $\pm\infty$, the result of an overflow, and NaN, "not a number", the result of an invalid operation such as $0/0$. The exponents of normal numbers therefore run from -126 to $+127$, that is from field 1 to field 254 [11].

Table 2 gives the widths. The width of the exponent field sets the range of a format. The width of the fraction sets its precision: `float32` carries about 7 significant decimal digits, `float64` about 16 [12]. Python's `float` and NumPy's default are `float64`. Datacenter GPUs such as the NVIDIA A100 and H100 and the AMD MI300 have full `float64` units and compute in it at about half their `float32` speed, which is what makes them usable for simulations and linear algebra. Consumer GPUs compute in `float64` at 1/32 to 1/64 of their `float32` speed. TPUs have no `float64` hardware. These ratios are those of the hardware generations of 2024 to 2026.

2.2 Decimal fractions in binary

The integer part of a number is converted by repeated division by 2: the remainders, read backwards, are the bits, so $141 \to 10001101_2$. The decimal part is converted by repeated doubling, each integer part produced being the next bit:

$$\begin{aligned} 0.92 \times 2 &= 1.84 \quad \text{bit 1, keep } 0.84 \\ 0.84 \times 2 &= 1.68 \quad \text{bit 1, keep } 0.68 \\ 0.68 \times 2 &= 1.36 \quad \text{bit 1, keep } 0.36 \\ 0.36 \times 2 &= 0.72 \quad \text{bit 0, keep } 0.72 \\ 0.72 \times 2 &= 1.44 \quad \text{bit 1, keep } 0.44 \end{aligned}$$

and so on, which gives $0.92 = 0.1110101110000101\ldots_2$. Listing 6 is the algorithm in four lines.

```
>>> x, bits = 0.92, ""
>>> for _ in range(16):
...     x *= 2; bits += str(int(x)); x -= int(x)
>>> bits
'1110101110000101'
```

Listing 6: The doubling algorithm: the first 16 fraction bits of 0.92.

Most decimal fractions have no finite binary expansion, as $1/3$ has none in base 10: the doubling never reaches 0, and the number is rounded to the nearest representable value when it

is read. Listing 7 shows the 64 bits of 1.92 as a `float64`. The module `struct` [13] converts between a value and its bytes: `struct.pack(">d", x)` gives the 8 bytes of `x`. The exponent field $01111111111_2 = 1023$ means 2^0. The fraction is the binary expansion of 0.92, cut after 52 bits, and its first 16 bits are those of Listing 6. Rounded to a `float32`, 1.92 becomes 1.9199999570846558, wrong in the 8th digit. The last line shows the best-known consequence of rounding on input. Each of 0.1, 0.2 and 0.3 is stored as the nearest `float64`, and the sum of the first two is not the stored 0.3 [14].

```
>>> import struct
>>> bits = format(struct.unpack(">Q", struct.pack(">d", 1.92))[0], "064b")
>>> bits[0], bits[1:12], bits[12:]
('0', '01111111111', '1110101111000010100011110101111000010100011110101111000')
>>> int(bits[1:12], 2) - 1023
0
>>> float(np.float32(1.92)) - 1.92
-4.2915344167226976e-08
>>> 0.1 + 0.2 == 0.3, 0.1 + 0.2
(False, 0.30000000000000004)
```

Listing 7: The 64 bits of 1.92, the first algae value of `C9.csv`, and two consequences of rounding.

2.3 Narrower formats for machine learning

Machine learning trades fraction bits for throughput, keeping the `float32` exponent where possible. Half the bytes means twice as many numbers per second through the memory system.

- bf16 [15] is 1 + 8 + 7 bits, the first 16 bits of a `float32`: the same range, 2^{16} times coarser, about 3 digits. It is the training format on TPUs [16] and on recent GPUs.
- fp16 [17] is 1 + 5 + 10 bits, with a range from 6×10^{-5} to 65 504, so overflow is a real risk.
- fp8, in the E4M3 and E5M2 variants [18], and the integer formats int8 and int4 with a scale factor per block serve the weights at inference.

Three digits are enough for the matrix products of training, the weights being kept in a `float32` master copy. A simulation that accumulates rounding errors over 10^9 steps needs `float64`.

3 Consequences of finite precision

Definition 3.1: Machine epsilon

The **machine epsilon** of a format is the spacing between 1 and the next representable number, 2^{-p} for a fraction of p bits: $2^{-23} \approx 1.2 \times 10^{-7}$ in `float32`, $2^{-52} \approx 2.2 \times 10^{-16}$ in `float64`. It is the relative precision of the format, since the spacing of representable numbers near x is $|x|$ times the machine epsilon.

Listing 8 shows the consequences of the formats for everyday code.

- Two floats are compared with a tolerance, never with `==`: `math.isclose` [19] for scalars, `np.allclose` [20] for arrays.
- Rounding errors accumulate. Adding 0.1 ten million times in `float32` gives 1 000 000.1. In `float64` the same sum gives 1 000 000.0. `np.finfo` prints the machine epsilon of Definition 3.1.

- The dtype of an array is a decision that trades memory and speed against precision. Measured data rarely justify more than `float32`. The arithmetic done on them often does.

For measured data with 4 significant digits, `float32` is the right format, not an overkill: it is the narrowest IEEE 754 format that keeps 4 digits, since fp16 and bf16 keep about 3. `float64` is the overkill for such data. The arithmetic done on the data may still need it.

```
>>> import math
>>> math.isclose(0.1 + 0.2, 0.3)
True
>>> np.sum(np.full(10**7, 0.1, dtype=np.float32)), np.sum(np.full(10**7, 0.1))
(np.float32(1000000.1), np.float64(1000000.0))
>>> np.finfo(np.float32).eps, np.finfo(np.float64).eps
(np.float32(1.1920929e-07), np.float64(2.220446049250313e-16))
```

Listing 8: Comparison with a tolerance, accumulated rounding error, and the machine epsilon of the two formats.

Goldberg's survey [21] remains the standard account of what the formats guarantee and what they do not.

Troubleshooting

Three symptoms point back to the formats of this chapter.

symptom	cause	remedy
A result differs in the last digits between two runs or two machines, by about 10^{-7} relative in `float32` or 10^{-16} in `float64`.	Rounding: the operations were done in a different order or by a different library (Section 3).	Compare with a tolerance, `math.isclose` or `np.allclose`, not with ==.
An integer array shows negative or wrapped values after a multiplication or a running sum.	Overflow of a fixed-width dtype, silent for arrays (Definition 1.2).	Use a wider dtype, `int64`, or a Python `int`.
A float array contains `inf`.	A value exceeded the largest number of the format, 3.4×10^{38} in `float32`, 65 504 in fp16 (Table 2).	Compute in `float64` and convert at the end.

Key points

- An n-bit pattern holds 2^n values, read as $N = \sum_i b_i 2^i$ (Equation (1)). A fixed-width integer wraps around modulo 2^n without any error message (Definition 1.2).
- The size of an array is the number of elements times the bytes per element. Binary prefixes are powers of 1024 and decimal prefixes powers of 1000, and they drift apart by 2.4% per step (Definition 1.3).
- A float is a sign, a biased exponent and a fraction (Definition 2.1). The exponent width sets the range and the fraction width the precision: 7 digits in `float32`, 16 in `float64`,

3 in bf16 and fp16 (Table 2).

- Most decimal fractions are rounded on input, so floats are compared with a tolerance and long sums are done in `float64` (Definition 3.1).

Exercises

Solutions are given from page 11.

A counter that turns negative

An event counter is stored as a NumPy `int32` and incremented 3×10^9 times. (a) What value does it hold at the end? (b) How many increments can an `int32` counter take before this happens, and what does `np.uint8(200) + np.uint8(100)` give?

The size of an image

A simulation writes images of 4096×4096 pixels. (a) How many bytes does one image take as `float64`, in MiB and in MB? (b) How many such images fit in the 128 GiB of RAM of the laptop of Section 1.3? (c) What changes with `float32`?

1.92 in sixteen bits

The `float32` pattern of 1.92 is `0 01111111 11101011100001010001111`. (a) Read off the bf16 value obtained by keeping the first 7 fraction bits and rounding to nearest. (b) Compare with the fp16 value, 1.919921875, and explain the difference in precision from Table 2. (c) Why are $0.1 + 0.2$ and 0.3 different as `float64`? Print both to 20 decimal places.

Solutions to the exercises

A counter that turns negative

(a) The largest `int32` is $2^{31} - 1 = 2\,147\,483\,647$. The counter wraps modulo 2^{32}, so it holds $3\,000\,000\,000 - 4\,294\,967\,296 = -1\,294\,967\,296$. (b) $2\,147\,483\,647$ increments. The next one gives $-2\,147\,483\,648$. For the unsigned sum, $300 - 256 = 44$.

Answer check

```
>>> np.array([3_000_000_000]).astype(np.int32)[0]
np.int32(-1294967296)
>>> np.uint8(200) + np.uint8(100)
np.uint8(44)
```

The second line also prints a `RuntimeWarning: overflow encountered in scalar add` with NumPy 2.

The size of an image

(a) $4096^2 \times 8 = 134\,217\,728$ bytes $= 128$ MiB $= 134.2$ MB. (b) 128 GiB/128 MiB $= 2^{10} = 1024$ images, with nothing left for the program itself. (c) Half: 64 MiB per image, 2048 images.

Answer check

```
>>> np.zeros((4096, 4096)).nbytes, np.zeros((4096, 4096), dtype=np.float32).
    nbytes
(134217728, 67108864)
>>> 128 * 2**30 // 134217728
1024
```

1.92 in sixteen bits

(a) The first 7 fraction bits are 1110101 and the next bit is 1, so the fraction rounds up to 1110110_2: $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{32} + \frac{1}{64} = 1.921875$, a relative error of 1×10^{-3}. (b) fp16 keeps 10 fraction bits, 2^3 times finer than bf16, hence 1.919921875, an error of 4×10^{-5}. bf16 keeps the 8 exponent bits instead, and with them the `float32` range. (c) Each of 0.1, 0.2 and 0.3 is rounded to the nearest `float64`. The rounded 0.1 and 0.2 both lie above their decimal values, and their sum rounds to the `float64` just above the rounded 0.3.

Answer check

```
>>> float(np.float16(1.92))
1.919921875
>>> [format(v, ".20f") for v in (0.1, 0.2, 0.3, 0.1 + 0.2)]
['0.10000000000000000555', '0.20000000000000001110',
 '0.29999999999999998890', '0.30000000000000004441']
```

References

[1] Bernd Blasius, Lars Rudolf, Guntram Weithoff, Ursula Gaedke, and Gregor F. Fussmann. Long-term cyclic persistence in an experimental predator–prey system. *Nature*, 577:226–230, 2020. Data: https://doi.org/10.6084/m9.figshare.10045976, CC BY 4.0.

[2] Bit. Wikipedia, https://en.wikipedia.org/wiki/Bit. accessed 20 October 2026.

[3] Word (computer architecture). Wikipedia, https://en.wikipedia.org/wiki/Word_(computer_architecture). accessed 20 October 2026.

[4] Python Software Foundation. Built-in functions: `bin()`. Python 3.12 documentation, https://docs.python.org/3/library/functions.html#bin. accessed 20 October 2026.

[5] Physical address extension. Wikipedia, https://en.wikipedia.org/wiki/Physical_Address_Extension. accessed 20 October 2026.

[6] NumPy developers. Data types. NumPy 2.5 user guide, https://numpy.org/doc/stable/user/basics.types.html. accessed 20 October 2026.

[7] Two's complement. Wikipedia, https://en.wikipedia.org/wiki/Two%27s_complement. accessed 20 October 2026.

[8] National Institute of Standards and Technology. Prefixes for binary multiples. https://physics.nist.gov/cuu/Units/binary.html. accessed 20 October 2026.

[9] NumPy developers. `numpy.ndarray.nbytes`. NumPy 2.5 reference, https://numpy.org/doc/stable/reference/generated/numpy.ndarray.nbytes.html. accessed 20 October 2026.

[10] IEEE. IEEE standard for floating-point arithmetic. IEEE Std 754-2019, https://standards.ieee.org/ieee/754/6210/, 2019. The 1985 edition introduced the single and double formats.

[11] Single-precision floating-point format. Wikipedia, https://en.wikipedia.org/wiki/Single-precision_floating-point_format. accessed 20 October 2026.

[12] Double-precision floating-point format. Wikipedia, https://en.wikipedia.org/wiki/Double-precision_floating-point_format. accessed 20 October 2026.

[13] Python Software Foundation. `struct`: interpret bytes as packed binary data. Python 3.12 documentation, https://docs.python.org/3/library/struct.html. accessed 20 October 2026.

[14] Python Software Foundation. Floating-point arithmetic: issues and limitations. Python 3.12 tutorial, https://docs.python.org/3/tutorial/floatingpoint.html. accessed 20 October 2026.

[15] bfloat16 floating-point format. Wikipedia, https://en.wikipedia.org/wiki/Bfloat16_floating-point_format. accessed 20 October 2026.

[16] Google Cloud. The bfloat16 numerical format. Cloud TPU documentation, https://cloud.google.com/tpu/docs/bfloat16. accessed 20 October 2026.

[17] Half-precision floating-point format. Wikipedia, https://en.wikipedia.org/wiki/Half-precision_floating-point_format. accessed 20 October 2026.

[18] Paulius Micikevicius et al. FP8 formats for deep learning. arXiv:2209.05433, https://arxiv.org/abs/2209.05433, 2022.

[19] Python Software Foundation. `math.isclose`. Python 3.12 documentation, https://docs.python.org/3/library/math.html#math.isclose. accessed 20 October 2026.

[20] NumPy developers. `numpy.allclose`. NumPy 2.5 reference, https://numpy.org/doc/stable/reference/generated/numpy.allclose.html. accessed 20 October 2026.

[21] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991. Reprint: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

Index

address, 2
address space, 2

bias, 5
binary prefixes, 3
bit, 1
byte, 1

decimal prefixes, 3
disk, 4
dtype, 2

exponent, 5

fraction, 4

machine epsilon, 7

RAM, 4
Registers, 2

sign bit, 4

two's complement, 3

word, 1
wrapped around, 3