Lecture 2 — Numbers in memory

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

Boris Bolliet

University of Cambridge

Outline

- Bits, bytes and integers; memory sizes
- Real numbers: IEEE 754 single and double precision
- Narrower formats in machine learning; consequences of finite precision
- Live demonstrations on the Lecture 1 data (`C9.csv`)

Key idea

A number is a fixed pattern of bits in memory; its width sets the range of the integers and the precision of the reals.

Bits and bytes

- A bit is 0 or 1; a byte (octet) is 8 bits, $2^8 = 256$ patterns.
- A word is the group of bits a processor handles as one unit: 64 bits = 8 bytes on current machines ("64-bit processor").
- Unsigned integer: $N = \sum_{i=0}^{n-1} b_i \, 2^i$.
- The 141 samples of `C9.csv`: $141 = 128 + 8 + 4 + 1 = 10001101_2$.

Key point

Memory is counted in bytes: `C9.csv` is 5 566 bytes; the lecture laptop has 128 GB of RAM and a 1 TB disk. A `float32` number occupies 32 bits = 4 bytes, a `float64` number 8 bytes; the processor's registers and memory addresses are 64 bits wide.

Binary number

✎ Integers in Python

```
>>> bin(141), int("10001101", 2), 0b10001101
('0b10001101', 141, 141)
>>> (141).bit_length(), (141).to_bytes(2, "big")
(8, b'\x00\x8d')
>>> import numpy as np
>>> np.array([141]).astype(np.int8)[0]
np.int8(-115)
>>> np.iinfo(np.int8), np.iinfo(np.uint8)
(min=-128, max=127), (min=0, max=255)
```

- A Python `int` has no size limit: `2**100` works. NumPy integer types have a fixed width, named by it: `int8` is a signed 8-bit integer, `uint8` an unsigned one; likewise `int32`, `int64` (the default).
- `uint8` holds 0 to 255; `int8` spends one bit on the sign (two's complement) and holds -128 to 127, so 141 does not fit and comes out as $141 - 256 = -115$, with no error (**wrap-around**).

bin() | NumPy data types

Memory sizes

Decimal unit	Bytes	Binary unit	Bytes
kB	10^3 B	KiB	2^{10} B
MB	10^6 B	MiB	2^{20} B
GB	10^9 B	GiB	2^{30} B
TB	10^{12} B	TiB	2^{40} B

$2^{10} = 1024$: 1 KiB = 1,024 B; 1 kB = 1,000 B. **B** means byte; **b** means bit: 100 Mb/s = 12.5 MB/s. Also, 4 TB = 4×10^{12} B $\approx$ 3.64 TiB.

Array size

Number of elements $\times$ bytes per element: a $10^4 \times 10^4$ float64 matrix uses $10^8 \times 8 = 800$ MB; float32 uses 400 MB.

NIST binary prefixes; Multiple-byte units

✎ Sizes on disk and in memory

```
$ ls -l C9.csv
-rw-r--r--  1 boris  staff  5566  8 Oct 07:04 C9.csv
$ du -h C9.csv
8.0K    C9.csv
$ python3.13 -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
```

- `ls` (list) `-l` (long format): size in bytes; `du` (disk usage) `-h` (human-readable): disk blocks, 4 KiB each here.
- The `float32` copy takes half the bytes (3 948 against 7 896) and keeps about 7 significant digits instead of 16.

ndarray.nbytes

Real numbers in binary

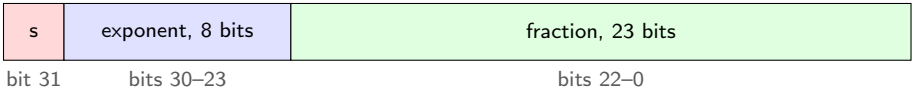
- Scientific notation in base 2: $x = (-1)^s \times 1.f \times 2^e$.
- IEEE 754 (1985, every CPU and GPU) stores s, e and f in fields of fixed width; e is stored with a constant added (the bias) so that the field is a positive integer.
- The fraction bits come from repeated doubling of the decimal part: $0.92 \to 1.84$ (1), $0.84 \to 1.68$ (1), $0.68 \to 1.36$ (1), $0.36 \to 0.72$ (0), $0.72 \to 1.44$ (1), ...: $0.92 = 0.11101\ldots_2$, so $1.92 = 1.1110101110000101\ldots_2$.
- The expansion does not end: most decimals are **not exactly representable**, like $1/3$ in base 10.

Key point

The exponent width fixes the range, the fraction width the precision.

Floating-point arithmetic (Python tutorial)

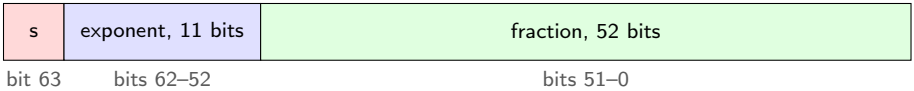
Single precision: fp32



- 32 bits = 4 bytes per number; value $= (-1)^s \, 1.f \, 2^{e-127}$.
- Bias $= 2^{k-1} - 1$ for a k-bit exponent field: 127 for 8 bits, 1023 for the 11 bits of fp64. The stored field is then an unsigned integer centred on 2^0, and $+0.0$ is the all-zero pattern.
- Field 1 to 254 gives 2^{-126} to 2^{127}; 0 and 255 are reserved for zero, subnormals, $\pm\infty$ and NaN.
- Range about 1.2×10^{-38} to 3.4×10^{38}; precision $2^{-23} \approx 1.2 \times 10^{-7}$, about 7 decimal digits.

Single-precision format

Double precision: fp64



- 64 bits = 8 bytes; bias 1023; exponent field 1 to 2046.
- Range about 2.2×10^{-308} to 1.8×10^{308}; precision $2^{-52} \approx 2.2 \times 10^{-16}$, about 16 digits.
- Python's `float` and NumPy's default `float64`.
- Datacenter GPUs (A100, H100) compute in `float64` at half their `float32` speed. Consumer GPUs do it at 1/32 to 1/64 of their `float32` speed. TPUs have no `float64`.

Key point

"32" and "64" count the bits per number.

Double-precision format

Narrower formats in machine learning

Format	s + e + f	Range	Digits
fp64	1 + 11 + 52	$10^{\pm 308}$	16
fp32	1 + 8 + 23	$10^{\pm 38}$	7
bf16	1 + 8 + 7	$10^{\pm 38}$	3
fp16	1 + 5 + 10	6×10^{-5} to 65 504	3
fp8 (E4M3)	1 + 4 + 3	up to 448	1–2

- bf16 is fp32 with 16 fraction bits dropped: same range, 2^{16} times coarser; the format of LLM training on TPUs and GPUs.
- Half the bytes: twice the numbers per second through memory (next section). Inference goes further: int8 and int4 weights.
- Enough precision for the matrix products of training (the weights stay in a `float32` copy); a simulation that accumulates over many steps needs `float64`.

✎ The first algae value, 1.92, in bits

```
>>> import struct, numpy as np
>>> 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)
```

- Stored exponent $01111111111_2 = 1023$; minus the bias 1023 gives 2^0: $1.92 = 1.92 \times 2^0$. The fraction is the binary expansion of 0.92, cut after 52 bits.
- Rounded to fp32, 1.92 is off by 4×10^{-8}: the 8th digit.

struct | numpy.finfo

Consequences of finite precision

```
>>> 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))
```

- Equality of floats is tested with a tolerance (`math.isclose`, `np.allclose`).
- Rounding errors accumulate over many operations; fp32 shows it after 10^7 additions.
- The `dtype` of an array is a choice: memory and speed against precision.

Goldberg, What every computer scientist should know about floating-point arithmetic

Summary

- A byte is 8 bits; n bits give 2^n values; kB versus KiB, b versus B.
- IEEE 754: sign, exponent, fraction; fp32 $\approx$ 7 digits, fp64 $\approx$ 16; equality with a tolerance.
- fp16, bf16 and fp8 trade digits for memory; the `dtype` of an array is a choice.
- Next: the machine, the shell and shell scripts.