

REVISED SECOND EDITION / 2026

No-Fear Math

for C/C++ Software Engineers

Understand the idea. Trust the code.

From reliable numbers to data and AI

30 VISUAL LESSONS • 4 WORKED PROJECTS

Ayman Alheraki

Mathematics you can put to work.

A clearer route from everyday calculations to dependable numerical code, data analysis and the foundations of modern AI.

Prepared by Ayman Alheraki

Revised Second Edition • October 2026

THE PROMISE

Start with a concrete question. Draw the relationship. Calculate one small example by hand. Then write code whose types, units and assumptions match the mathematics. This is the learning rhythm of the book.

You need ordinary programming experience: variables, conditions, loops, functions and arrays. You do not need a university mathematics sequence. Symbols are introduced at the moment they become useful.

WHAT CHANGED

The first C/C++ Engineering Edition offered a wide map of 49 topics. This revision turns that material into 30 connected lessons, four complete projects and four focused specialty guides. Repeated generic advice has been replaced by worked reasoning, declared input contracts and exercises with explanations.

The data and AI route now develops scaling, losses, gradients, a neuron, vector retrieval, attention and quantization. Advanced specialties are signposted so they do not interrupt the foundation.

ABOUT THE EXAMPLES

C examples use C11; C++ examples use C++20. Small fixed datasets keep every step inspectable. Each listing is a complete program. Described domains and target assumptions are part of the example, not optional footnotes.

Editorial note: this revision and its artwork were developed with AI assistance. The accompanying code was compiled and exercised in the verification environment recorded on page 95. The examples teach reasoning; domain-specific production systems need validation appropriate to their inputs and requirements.

A MESSAGE FROM AYMAN ALHERAKI

Why I prepared this book.

I originally prepared this book for myself. While developing low-level programming tools for my MyLang Toolchain project, MyLangTC.org, I needed to revisit important mathematical concepts, understand them precisely, and apply them confidently to real engineering problems.

That project has already taken substantial steps forward. I have developed MyLangAssembler and a new intermediate language called Structured Assembly, positioned between C and traditional assembly. I am designing it to serve as the intermediate-representation layer in my own toolchain, as an alternative to adopting the approaches offered by LLVM IR and MLIR. MyLangAssembler currently targets x86-64 processors on Windows and Linux. It has reached the alpha stage, and I plan to make it available soon for those who would like to try it.

My need for a clearer mathematical foundation led to the first edition. I read it carefully, benefited from it, and decided to share it so that others could benefit too. Its reception exceeded my expectations. Many experienced readers examined it, identified errors, offered generous encouragement, and helped me see how it could become better. I value both their corrections and their support.

Their response encouraged me to prepare this second edition with greater clarity, tighter focus, and particular attention to data and artificial intelligence through C and C++. The purpose remains practical: to understand the mathematics well enough to use it, question it, and judge the code that implements it.

I want to be entirely clear about how this work is produced: I rely extensively on artificial intelligence to suggest material, develop explanations and examples, organize the content, and help design the book. I describe my role as preparing, assembling, and reviewing this material. I have never claimed to have authored it independently of these tools. Selecting, connecting, checking, and revising the material still requires substantial effort. When I have not reviewed a book sufficiently, I label it Draft.

Pointing out that AI was involved therefore reveals nothing I have concealed. Specific corrections and thoughtful criticism are welcome; dismissive labels and speculation about my motives do not improve a formula, an example, or a reader's understanding. I share this work to advance my projects and help others learn, not to seek fame. Readers who reject AI-assisted learning can choose other books and stop following my accounts or visiting my website. There is no need to turn that choice into personal hostility.

To everyone who wants to explore the remarkable possibilities these tools offer, learn alongside me, and contribute to making the material better: you are warmly welcome. You have my appreciation and respect. This book is for those who want to understand, build, and move forward together.

Ayman Alheraki

Preparer of this book | MyLangTC.org

Read. Predict. Run. Explain.

01 READ THE PICTURE

The first page of each lesson develops one mental model. Read the units and the assumptions before the formula.

02 PREDICT THE SMALL EXAMPLE

Work through the numbered steps before looking at the program output. A hand-checkable case is your first independent oracle.

03 RUN THE COMPLETE PROGRAM

Every lesson has one source file in the companion examples folder. Open the corresponding file and build it using the language standard shown below.

04 CHANGE ONE THING

Change an endpoint, scale, size or sign. Explain the expected effect before running again. Use the checkpoint answers on pages 84-88 afterward.

BUILD ONE C EXAMPLE

```
gcc -std=c11 -O2 examples/lesson_01.c -lm -o lesson_01
```

BUILD ONE C++ EXAMPLE

```
g++ -std=c++20 -O2 examples/lesson_02.cpp -o lesson_02
```

Run the produced program from your IDE, or use `./lesson_01` on Linux/macOS and `lesson_01.exe` on Windows. On a compiler toolchain that does not need a separate math-library option, omit `-lm`. A matching C11/C++20 implementation and standard library are required.

USING CMAKE OR CLION

Open the companion CMakeLists.txt. Each lesson and project has its own executable target, so you run one complete program at a time. CTest runs output checks and boundary tests. No external numerical packages or downloaded datasets are needed.

The PDF keeps text and code selectable. For exact whitespace, use the companion source files. Expected decimal results assume the ordinary binary64 environment used during verification; final printed digits may differ on other floating-point implementations.

CONTENTS / 1 OF 2

Choose your next idea.

01	Give every number a meaning	8
02	Turn a formula into a function	10
03	Read growth with powers and logs	12
04	See bits, then write the condition	14
05	Check the range before multiplying	16
06	Reason about cycles and scales	18
07	Floating point is a moving ruler	20
08	Spend an explicit error budget	22
09	A sum is a loop with a name	24
10	Prove a loop in one sentence	26
11	Predict work before measuring time	28
12	Use sets and graphs to expose structure	30
13	Count outcomes before using Bayes	32
14	Keep the shape of the data	34
15	Measure uncertainty, not just speed	36

A FAST FOUNDATION

Read Lessons 1-12 in order. Each lesson uses earlier ideas: units lead to range checks, range checks lead to trustworthy loops, and loops lead to complexity.

Each lesson takes two pages: understand the idea, then apply it. The numbering in this contents list matches the visible PDF page number.

CONTENTS / 2 OF 2

Choose your next idea.

16	Trade memory for a known error rate	38
17	A vector is a list with geometry	40
18	Check matrix shapes before values	42
19	Keep coordinates in their frames	44
20	A derivative is local sensitivity	46
21	Train a model by reducing a loss	48
22	Accumulation turns rates into totals	50
23	Signals are weighted neighborhoods	52
24	Count bytes as carefully as operations	54
25	Parallel work still has a serial part	56
26	Prepare data without leaking the answer	58
27	Turn scores into a stable loss	60
28	Backpropagation is the chain rule reused	62
29	Similarity and attention share a dot product	64
30	Make quantization an error-budget decision	66
	Project A / Latency report	68
	Project B / Image filter in C	71
	Project C / Regression	74
	Project D / Vector retrieval	77
	Specialty guides	80
	Checkpoint answers	84
	Formula atlas	89
	Glossary	92
	References and editorial notes	94

A tiny language for big ideas.

Mathematical notation becomes easier when you translate each symbol into an action you already know from code. Keep this page nearby for the first few lessons.

X[I]

Read component i of an ordered collection.

SUM OF X[I]

Loop over the chosen indices and add the values.

|X|

Take the magnitude: discard the sign of a real number.

SQRT(X)

Find the nonnegative number whose square is x ; require $x \geq 0$.

X² IN MATHEMATICS

Multiply x by itself. In C/C++, write $x*x$, not x^2 .

[A,B]

Include both endpoints.

[A,B)

Include a , exclude b . This matches common iterator ranges.

F(X)

Apply a rule to an input x .

P(A|B)

Count the fraction of B outcomes that also belong to A .

DL/DW

Ask how loss L changes locally when weight w changes.

A^T

Transpose a matrix: exchange its row and column indices.

APPROXIMATELY EQUAL

An approximation with an error scale or stated model, not exact identity.

LESSON 01 / UNDERSTAND

Give every number a meaning

A number becomes useful when you know what it measures.

Before allocating a buffer or estimating a transfer, write the units. This catches mistakes that a compiler cannot see.

A quantity has three parts: a value, a unit and a role. The number 48,000 could mean audio frames per second, bytes, or samples. Those meanings are not interchangeable.

A ratio compares quantities; a rate divides one kind of quantity by another. Bytes divided by bytes has no unit. Bytes divided by seconds is a throughput.

Decimal MB means 1,000,000 bytes. Binary MiB means 1,048,576 bytes. A C byte is `sizeof(char)`; it is not universally eight bits. The mainstream targets assumed by the audio example use eight-bit bytes and four-byte float.

$$\begin{array}{ccccccc} \mathbf{48,000} & \times & \mathbf{2} & & \times & \mathbf{4} & = & \mathbf{384,000} \\ \text{frames} & & \text{samples / frame} & & & \text{bytes / sample} & & \text{bytes} \end{array}$$

The units cancel; the meaning stays visible.

WORK IT OUT

1. One second of stereo audio has 48,000 frames, with two samples in each frame.
2. Each sample occupies four bytes: $48,000 \times 2 \times 4 = 384,000$ bytes.
3. $384,000 / 1,048,576 = 0.3662109375$ MiB. The smaller-looking number describes the same memory.

frames x samples/frame x bytes/sample = bytes

LESSON 01 / C11

From idea to code.

Companion file: `examples/lesson_01.c`

```
#include <limits.h>
#include <stdio.h>
int main(void) {
    _Static_assert(CHAR_BIT == 8, "8-bit bytes required");
    _Static_assert(sizeof(float) == 4, "4-byte float required");
    const size_t frames = 48000;
    const size_t channels = 2;
    const size_t bytes = frames * channels * sizeof(float);
    printf("%zu bytes\n", bytes);
    printf("%.9f MiB\n", (double)bytes / 1048576.0);
    return 0;
}
```

EXPECTED OUTPUT

384000 bytes
0.366210938 MiB

The cast makes division floating-point. The constants are small enough for this target; arbitrary sizes need the checked multiplication in Lesson 5.

WATCH THE BOUNDARY

Going from 100 ms to 80 ms is a 20% time reduction, but throughput rises by 25% if work per run is fixed. Always name the denominator.

TRY IT BEFORE READING THE ANSWER

A file contains 750 MiB. At an ideal 2.5 Gbit/s, how many seconds does the payload take? Ignore every protocol overhead.

Answer 01 • page 84

LESSON 02 / UNDERSTAND

Turn a formula into a function

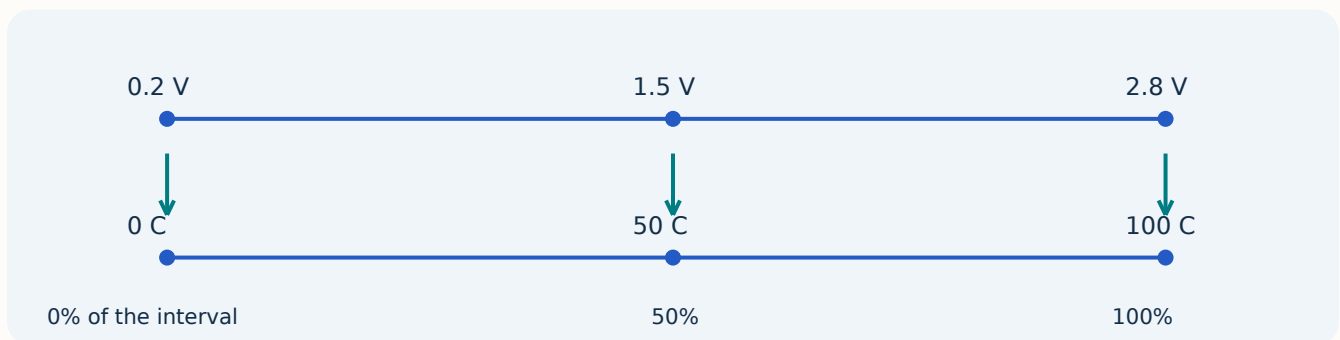
A formula is a small program with a contract.

Calibration, UI sliders and feature scaling all map one interval onto another. Derive the mapping from the endpoints.

A function has an input domain and an output range. State them before choosing a type. A sensor calibration may accept only a particular voltage interval.

First convert the input into a fraction of its interval. Then use that fraction to travel between the output endpoints. Linear interpolation assumes a straight-line relationship.

Clamping changes the contract: it replaces out-of-range values by endpoints. That is useful for a slider, but it can hide a failed sensor. The example rejects invalid input instead.



WORK IT OUT

1. A sensor gives 0.2 V at 0 C and 2.8 V at 100 C.
2. At 1.5 V, the fraction is $(1.5 - 0.2) / (2.8 - 0.2) = 0.5$.
3. The output is $0 + 0.5 \times (100 - 0) = 50$ C. Check both endpoints as well as this midpoint.

$$t = (x - a) / (b - a); y = low + t(high - low)$$

LESSON 02 / C++20

From idea to code.

Companion file: `examples/lesson_02.cpp`

```
#include <cmath>
#include <iostream>
#include <stdexcept>
double celsius(double volts) {
    if (!std::isfinite(volts) || volts < 0.2 || volts > 2.8)
        throw std::domain_error("volts outside calibration");
    const double t = (volts - 0.2) / (2.8 - 0.2);
    return std::lerp(0.0, 100.0, t);
}
int main() {
    std::cout << celsius(1.5) << '\n';
}
```

EXPECTED OUTPUT

50

C++20 `std::lerp` expresses the interpolation directly. For a general mapping, reject equal source endpoints and define how reversed endpoints behave.

WATCH THE BOUNDARY

An exact equality check is appropriate for exactly equal endpoints. A near-zero interval needs a separate, unit-based conditioning rule; a universal epsilon is not a solution.

TRY IT BEFORE READING THE ANSWER

Map $x = 25$ from $[10, 70]$ to $[1000, 4000]$. What happens if the caller supplies $x = 80$?

Answer 02 • page 84

LESSON 03 / UNDERSTAND

Read growth with powers and logs

A logarithm asks how many times you multiply.

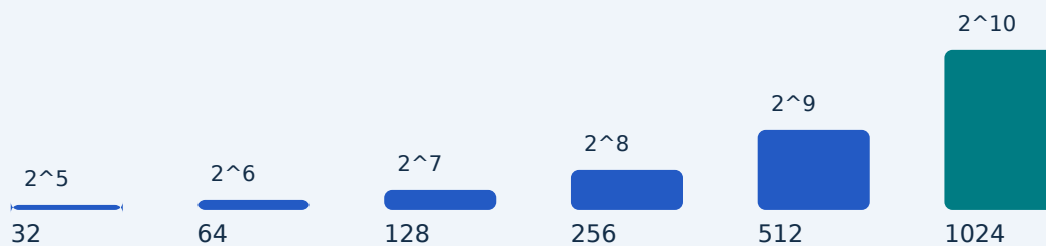
Powers count states. Logarithms recover the number of decisions, doublings or bits needed to reach them.

Two choices per bit give 2^b possible bit patterns. Ten bits distinguish 1,024 patterns, with unsigned values from 0 through 1,023.

$\log_2(1024) = 10$ because $2^{10} = 1024$. You do not need a floating-point logarithm to count the bits of an integer.

For N distinct states numbered 0 through $N-1$, the mathematical bit count is $\text{ceil}(\log_2(N))$ when N is positive. A single state needs zero information bits, although a stored object still occupies space.

Each step doubles the capacity.

**WORK IT OUT**

1. A table needs capacity for 1,000 elements. The adjacent powers of two are 512 and 1,024.
2. A capacity of 1,024 is the first one that fits.
3. Before doubling an integer, check that the next value is representable. A shift by the type width is not a valid shortcut.

$$2^{10} = 1024; \log_2(1024) = 10$$

LESSON 03 / C++20

From idea to code.

Companion file: `examples/lesson_03.cpp`

```
#include <cstdint>
#include <iostream>
#include <limits>
#include <optional>
std::optional<std::size_t> next_power(std::size_t n) {
    std::size_t p = 1;
    while (p < n) {
        if (p > std::numeric_limits<std::size_t>::max() / 2)
            return std::nullopt;
        p *= 2;
    }
    return p; // policy: zero requests get capacity one
}
int main() {
    const auto p = next_power(1000);
    if (p) std::cout << *p << '\n';
}
```

EXPECTED OUTPUT

1024

This loop uses at most the number of value bits in `size_t`. It makes the overflow check visible; C++20 bit utilities offer alternatives with their own preconditions.

WATCH THE BOUNDARY

In C and C++, `^` means bitwise XOR, not exponentiation. Use multiplication for a small exact power and a suitable math function for a real exponent.

TRY IT BEFORE READING THE ANSWER

How many bits distinguish one million IDs? What capacity is the next power of two at least 5,000?

Answer 03 • page 84

LESSON 04 / UNDERSTAND

See bits, then write the condition

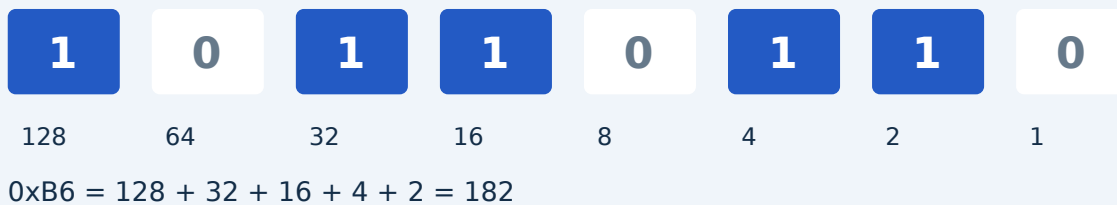
Bitwise logic works on positions; Boolean logic works on truth.

Flags and predicates become easier to review when the intended rule is written in a sentence first.

A binary numeral is a sum of powers of two. Hexadecimal groups four bits into one digit. 0xB6 is 1011 0110, or $128 + 32 + 16 + 4 + 2 = 182$.

Bitwise AND selects shared set bits. OR combines set bits. XOR marks differences. Logical && and || combine truth values and short-circuit evaluation.

De Morgan gives a useful reading aid: not(A and B) means not A or not B. Keep predicates free of side effects when you rearrange them.

**WORK IT OUT**

1. A file may be writable only when the user is authenticated and is either the owner or an administrator.
2. Write `authenticated && (owner || admin)`. The parentheses expose the grouping.
3. Three Boolean inputs have eight combinations. Checking all eight is a small, complete truth-table test.

NOT (A AND B) = (NOT A) OR (NOT B)

LESSON 04 / C11

From idea to code.

Companion file: `examples/lesson_04.c`

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
bool can_write(bool auth, bool owner, bool admin) {
    return auth && (owner || admin);
}
int main(void) {
    const uint32_t read_flag = UINT32_C(1) << 1;
    const uint32_t write_flag = UINT32_C(1) << 3;
    const uint32_t flags = read_flag | write_flag;
    printf("write flag: %d\n", (flags & write_flag) != 0);
    printf("allowed: %d\n", can_write(true, false, true));
    return 0;
}
```

EXPECTED OUTPUT

```
write flag: 1
allowed: 1
```

The shift counts are fixed and valid. Fixed-width types such as `uint32_t` exist only on implementations with the corresponding exact-width type.

WATCH THE BOUNDARY

Use `p != NULL && *p > 0` for short-circuit pointer checking. Replacing `&&` with `&` evaluates both operands and destroys that protection.

TRY IT BEFORE READING THE ANSWER

Write `0xA7` in binary. Rewrite `!(ready && (cached || local))` without a negation around a compound expression.

Answer 04 • page 84

LESSON 05 / UNDERSTAND

Check the range before multiplying

A valid mathematical answer may not fit the machine type.

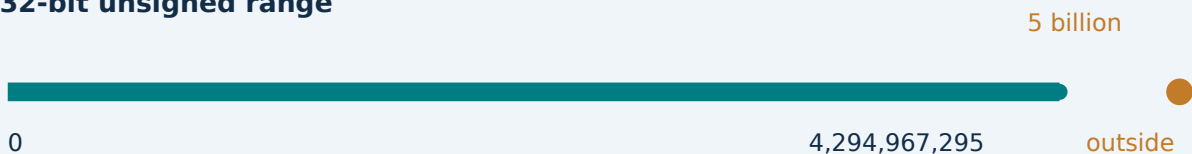
Allocation sizes, image dimensions and counters need a range argument as well as a formula.

Unsigned arithmetic wraps modulo one more than its maximum value. That is useful for some counters but dangerous for a byte count. Signed integer overflow has undefined behavior.

A conversion after an operation is too late to enlarge its intermediate type. Widen before multiplication, then still ask whether the wider product fits.

For nonnegative integers, $a*b$ fits in a maximum M exactly when a is zero or $b \leq M/a$. This transforms a dangerous multiplication into a safe comparison.

32-bit unsigned range



Check the operands before forming the product.

WORK IT OUT

1. A 32-bit unsigned type holds at most 4,294,967,295.
2. $100,000 \times 50,000$ is 5,000,000,000, so the product does not fit.
3. On a 64-bit `size_t` target it fits, but the same API should also reject inputs whose product exceeds that target's actual limit.

$a*b$ fits when $a == 0$ OR $b \leq \text{MAX}/a$

LESSON 05 / C11

From idea to code.

Companion file: `examples/lesson_05.c`

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
bool size_product(size_t a, size_t b, size_t *out) {
    if (out == NULL || (a != 0 && b > SIZE_MAX / a))
        return false;
    *out = a * b;
    return true;
}
int main(void) {
    size_t bytes = 0;
    if (!size_product(48000, 8, &bytes)) return 1;
    printf("%zu\n", bytes);
    printf("rejected: %d\n", !size_product(SIZE_MAX, 2, &bytes));
    return 0;
}
```

EXPECTED OUTPUT

```
384000
rejected: 1
```

The output pointer is checked, and failed calls do not modify the result. A dimensions product with three factors needs two checked multiplications.

WATCH THE BOUNDARY

An unchecked cast from negative int to `size_t` can create a huge value. Validate the signed input before converting it. Small unsigned operands may undergo integer promotion.

TRY IT BEFORE READING THE ANSWER

Give safe preconditions for adding two `size_t` values and for rounding an integer `x` up to an alignment `A`.

Answer 05 • page 84

LESSON 06 / UNDERSTAND

Reason about cycles and scales

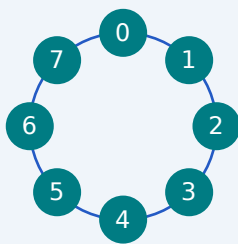
A remainder describes a cycle; a scale describes a stored fraction.

Circular buffers and fixed-point values look like ordinary integers, but each has a different mathematical meaning.

For a ring with capacity 8, the positions are 0 through 7. The next position after 7 is 0. A modulo operation expresses this wrap, provided the capacity is positive.

C remainder follows truncated division: $-1 \% 8$ is -1 . For a positive modulus m , compute $r = x \% m$ and add m only if r is negative to obtain a nonnegative remainder.

A fixed-point raw integer q with F fractional bits means $q / 2^F$. Q8.8 raw 384 means 1.5. Multiplication needs a wider intermediate, explicit rounding and an output-range check.



7 -> 0

Position wraps; capacity remains 8.

Modulo describes a circle of states.

WORK IT OUT

1. A 12 ms task and an 18 ms task first repeat their shared phase after 36 ms.
2. Euclid gives $\text{gcd}(12, 18) = 6$. Then $\text{lcm} = (12/6) \times 18 = 36$.
3. Divide before multiplying to reduce risk; the remaining product still needs a range check for arbitrary periods.

$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$; $\text{real} = \text{raw} / 2^F$

LESSON 06 / C11

From idea to code.

Companion file: `examples/lesson_06.c`

```
#include <stddef.h>
#include <stdio.h>
size_t next_ring(size_t i, size_t capacity) {
    /* Precondition: capacity > 0 and i < capacity. */
    return i == capacity - 1 ? 0 : i + 1;
}
unsigned gcd_u(unsigned a, unsigned b) {
    while (b != 0) {
        unsigned r = a % b;
        a = b; b = r;
    }
    return a;
}
int main(void) {
    printf("next: %zu\n", next_ring(7, 8));
    printf("gcd: %u\n", gcd_u(12, 18));
    printf("scaled: %.1f\n", 384 / 256.0);
    return 0;
}
```

EXPECTED OUTPUT

```
next: 0
gcd: 6
scaled: 1.5
```

The ring function avoids a potentially overflowing `i+1` at the boundary. Its precondition is explicit; this small internal helper assumes validation by its caller.

WATCH THE BOUNDARY

A mask replaces modulo only for a power-of-two capacity. Also, a right shift of a negative signed value is not a portable C11 rounding policy; division truncates toward zero.

TRY IT BEFORE READING THE ANSWER

Compute `gcd(252,105)`. For an 8-bit wrapping tick counter, what elapsed value does 4-250 represent if fewer than 256 ticks passed?

Answer 06 • page 84

LESSON 07 / UNDERSTAND

Floating point is a moving ruler

Floating-point spacing changes with the size of the number.

Understanding the representation explains many surprising results without memorizing special cases.

A floating-point value stores a sign, a significand and an exponent. Think of a ruler whose marks get farther apart as its exponent grows.

On common IEEE binary64 targets, double has 53 bits of significand precision. Every integer up to 2^{53} is exactly representable; above that, some adjacent integers are not.

Decimal 0.1 repeats in binary, just as $1/3$ repeats in decimal. Rounding is unavoidable. Infinity and NaN exist on IEEE implementations; test your target rather than assuming every C or C++ implementation has the same format.

Near a smaller magnitude



At twice the magnitude



Illustrative spacing: gaps grow with the exponent.

WORK IT OUT

1. 0.1 and 0.2 are each rounded when stored.
2. Their sum is rounded again. On binary64 it is typically 0.30000000000000004.
3. The separately rounded value 0.3 is different. This is a representation effect, not a broken addition operator.

local spacing near 1 on binary64 = 2^{-52}

LESSON 07 / C++20

From idea to code.

Companion file: `examples/lesson_07.cpp`

```
#include <cmath>
#include <iomanip>
#include <iostream>
#include <limits>
int main() {
    static_assert(std::numeric_limits<double>::is_iec559);
    static_assert(std::numeric_limits<double>::digits == 53);
    const double x = 0.1 + 0.2;
    const double after = std::nextafter(1.0, 2.0);
    std::cout << std::setprecision(17) << x << '\n';
    std::cout << "equal: " << (x == 0.3) << '\n';
    std::cout << "spacing: " << after - 1.0 << '\n';
}
```

EXPECTED OUTPUT

```
0.300000000000000004
equal: 0
spacing: 2.2204460492503131e-16
```

The static assertions make the demonstrated representation a build requirement. `nextafter` asks for the next representable value toward its second argument.

WATCH THE BOUNDARY

Epsilon is the gap near one, not a universal measurement tolerance. Money, timestamps and noisy sensor values each require different error contracts.

TRY IT BEFORE READING THE ANSWER

Can a binary64 double distinguish the integers 9,007,199,254,740,992 and 9,007,199,254,740,993?

[Answer 07](#) • [page 85](#)

LESSON 08 / UNDERSTAND

Spend an explicit error budget

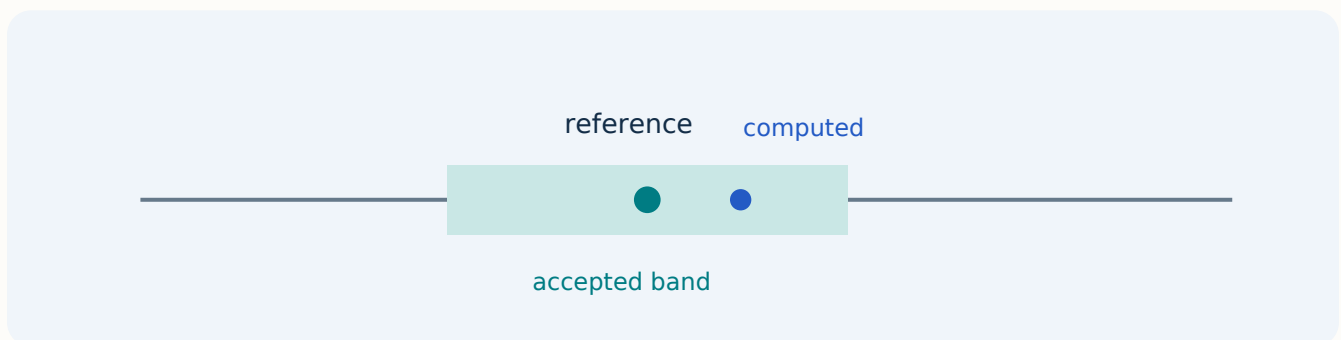
Choose accuracy from the problem, then choose the calculation.

Good numerical code controls error instead of demanding impossible exactness from every real-valued calculation.

Absolute error measures distance in the output unit. Relative error divides by the magnitude of a nonzero reference. Near zero, relative error alone is unhelpful.

Conditioning describes sensitivity of the problem. Stability describes how much extra error the algorithm introduces. A stable algorithm cannot recover information missing from uncertain input.

Subtraction can expose rounding error when nearly equal inputs cancel. For small x , $\log1p(x)$ and $\expm1(x)$ evaluate $\log(1+x)$ and $\exp(x)-1$ without first discarding tiny increments.

**WORK IT OUT**

1. A reading of 100.02 C compared with 100.00 C has absolute error 0.02 C.
2. Its relative error is 0.0002, or 0.02%. Either may matter, depending on the specification.
3. A combined rule accepts a difference below $\max(\text{absolute tolerance}, \text{relative tolerance} \times \text{scale})$. Decide those tolerances in physical units and meaningful fractions.

accept when $|a-b| \leq \max(\text{abs_tol}, \text{rel_tol} \times \max(|a|, |b|))$

LESSON 08 / C++20

From idea to code.

Companion file: `examples/lesson_08.cpp`

```
#include <algorithm>
#include <cmath>
#include <iostream>
bool near(double a, double b, double at, double rt) {
    if (!std::isfinite(a) || !std::isfinite(b) ||
        !std::isfinite(at) || !std::isfinite(rt) ||
        at < 0 || rt < 0) return false;
    const double scale = std::max(std::abs(a), std::abs(b));
    if (scale == 0) return true;
    // Scale first: avoid overflowing a-b for opposite signs.
    return std::abs(a/scale - b/scale) <=
        std::max(at/scale, rt);
}
int main() {
    std::cout << near(0.1 + 0.2, 0.3, 1e-12, 1e-12) << '\n';
}
```

EXPECTED OUTPUT

1

This comparator accepts only finite values and finite nonnegative tolerances. It deliberately rejects infinities and NaNs. It is an approximate numerical test, not exact arithmetic at the tolerance boundary.

WATCH THE BOUNDARY

Approximate equality need not be transitive. With tolerance 0.1, 0 is near 0.09 and 0.09 is near 0.18, but 0 is not near 0.18. Do not use it as container key equality.

TRY IT BEFORE READING THE ANSWER

Which is usually better near $x = 0$: $\log(1+x)$ or $\log_{1p}(x)$? Can more precision fix a poorly measured input?

[Answer 08](#) • [page 85](#)

LESSON 09 / UNDERSTAND

A sum is a loop with a name

Sigma notation is repeated addition, not a new operation.

Running totals and range queries become simple when the boundary convention is clear.

The sum of $x[i]$ for i from 0 through $n-1$ is exactly what a loop with $i < n$ computes. The notation just hides the bookkeeping.

A prefix array P starts with $P[0] = 0$. $P[k]$ stores the sum of the first k elements. Then the sum of a half-open interval $[l, r)$ is $P[r] - P[l]$.

A wider integer accumulator postpones overflow but does not remove its possibility. Floating-point sums are also order-sensitive; pairwise or compensated summation can reduce error under appropriate conditions.



WORK IT OUT

1. For $x = [3, 1, 4, 2]$, write $P = [0, 3, 4, 8, 10]$.
2. The interval $[1, 3)$ includes 1 and 4. Its sum is $P[3] - P[1] = 8 - 3 = 5$.
3. Build once in $O(n)$, then answer each valid range query in $O(1)$. The extra storage is $n+1$ totals.

$$P[0] = 0; P[k+1] = P[k] + x[k]$$

LESSON 09 / C11

From idea to code.

Companion file: `examples/lesson_09.c`

```
#include <stdio.h>
int main(void) {
    const int x[4] = {3, 1, 4, 2};
    long long p[5] = {0};
    for (size_t i = 0; i < 4; ++i)
        p[i + 1] = p[i] + x[i];
    const size_t left = 1, right = 3;
    printf("range sum: %lld\n", p[right] - p[left]);
    return 0;
}
```

EXPECTED OUTPUT

range sum: 5

The fixed data keep every total small. A reusable version must check $l \leq r \leq n$ and ensure every accumulated total and subtraction fits.

WATCH THE BOUNDARY

The identity $1 + \dots + n = n(n+1)/2$ does not license an overflowing intermediate. Divide an even factor first and still check the multiplication.

TRY IT BEFORE READING THE ANSWER

Using the same prefix array, compute $[0,4)$, $[2,2)$ and $[2,4)$. What does the empty interval mean?

Answer 09 • page 85

LESSON 10 / UNDERSTAND

Prove a loop in one sentence

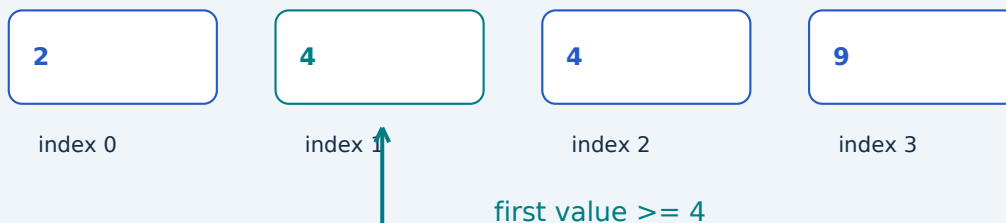
An invariant says what stays true while the loop makes progress.

A short invariant often prevents more bugs than a long list of special-case patches.

A proof needs a starting fact, a preservation step and a reason to stop. For a maximum scan, the fact is: best is the maximum of the prefix already visited.

For lower-bound search, keep $[lo, hi)$ as the undecided region. Every earlier element is less than the key; every element at or after hi is at least the key.

The interval shrinks each iteration. When it is empty, lo is the first position whose value is at least the key, or n if there is no such value.

**WORK IT OUT**

1. Search $[2, 4, 4, 9]$ for 4. Start $lo=0$, $hi=4$.
2. $mid=2$ has value 4, so $hi=2$. Then $mid=1$ also has 4, so $hi=1$.
3. $mid=0$ has 2, so $lo=1$. The loop ends at index 1, the first 4.

$0 \leq lo \leq hi \leq n$; $mid = lo + (hi-lo)/2$

LESSON 10 / C++20

From idea to code.

Companion file: `examples/lesson_10.cpp`

```
#include <cstdint>
#include <iostream>
#include <span>
std::size_t lower(std::span<const int> a, int key) {
    std::size_t lo = 0, hi = a.size();
    while (lo < hi) {
        const std::size_t mid = lo + (hi - lo) / 2;
        if (a[mid] < key) lo = mid + 1;
        else hi = mid;
    }
    return lo;
}
int main() {
    const int a[] = {2, 4, 4, 9};
    std::cout << lower(a, 4) << '\n';
}
```

EXPECTED OUTPUT

1

The precondition is sorted nondecreasing input. The midpoint form avoids adding the two endpoints. Empty arrays naturally return zero without a dereference.

WATCH THE BOUNDARY

Tests demonstrate selected cases; they do not prove a universal claim. A proof about mathematical integers is incomplete if the implementation can overflow or access a dead object.

TRY IT BEFORE READING THE ANSWER

What should `lower` return for keys 1, 5 and 10 on the same array? State a maximum-scan invariant.

Answer 10 • page 85

LESSON 11 / UNDERSTAND

Predict work before measuring time

Count the structure of the work, then measure its cost.

Complexity helps choose an algorithm. It does not predict an exact number of milliseconds.

A linear scan visits n items. Trying every unordered pair visits $n(n-1)/2$ pairs. Twelve independent switches create $2^{12} = 4,096$ configurations.

O is an asymptotic upper bound; Θ is a matching upper and lower bound. Worst-case, expected and amortized are different qualifiers, not synonyms.

Merge sort has about $\log_2(n)$ levels with $O(n)$ merge work per level. A balanced search tree has logarithmic height; a poorly shaped binary search tree can have linear height. A heap is partially ordered, not fully sorted.

Input n	Linear visits	Unordered pairs
1,000	1,000	499,500
10,000	10,000	49,995,000

10x the input means about 100x as many pairs.

WORK IT OUT

1. For $n=1,000$, all unordered pairs number 499,500.
2. For $n=10,000$, the count becomes 49,995,000, about 100 times as many.
3. A scan grows only tenfold. Changing the algorithm can dominate optimizing each pair operation.

unordered pairs = $n(n-1)/2$; merge work $\sim n \log_2(n)$

LESSON 11 / C11

From idea to code.

Companion file: `examples/lesson_11.c`

```
#include <stdio.h>
unsigned levels(unsigned n) {
    unsigned count = 0;
    while (n > 1) {
        n = n / 2 + n % 2; /* ceiling division by 2 */
        ++count;
    }
    return count;
}
int main(void) {
    printf("levels for 1000: %u\n", levels(1000));
    printf("pairs for 1000: %llu\n", 1000ULL * 999ULL / 2);
    return 0;
}
```

EXPECTED OUTPUT

```
levels for 1000: 10
pairs for 1000: 499500
```

$n/2 + n\%2$ rounds upward without risking overflow in $n+1$. The pair example is a bounded calculation, not a general combinatorics implementation.

WATCH THE BOUNDARY

Geometric vector growth gives amortized constant-time append because total moved elements form a geometric sum. It does not mean each individual element is moved only once.

TRY IT BEFORE READING THE ANSWER

Choose 3 distinct codecs from 10. How many choices if order is irrelevant? If order defines a pipeline?

Answer 11 • page 85

LESSON 12 / UNDERSTAND

Use sets and graphs to expose structure

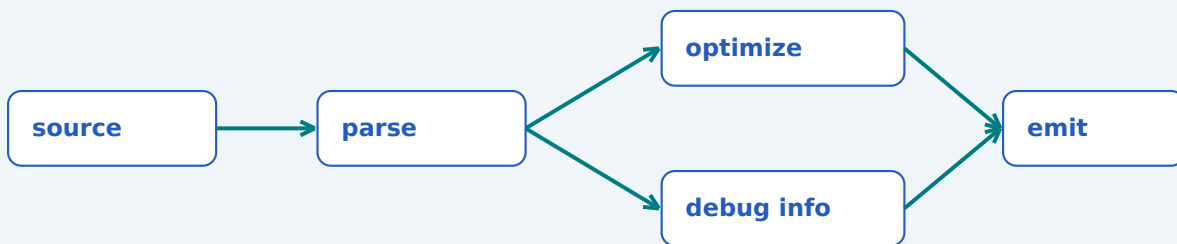
A set answers membership; a graph answers connection.

Dependency tools, data pipelines and compilers use the same small family of structures.

Union combines memberships; intersection keeps shared members; difference removes members. Equal keys in a hash container must produce equal hash values.

A graph consists of vertices and edges. Direction matters: $A \rightarrow B$ may mean B depends on A. State that convention, because reversing it reverses a build order.

Breadth-first search finds shortest edge counts in an unweighted graph. Dijkstra needs nonnegative weights. A directed acyclic graph has a topological order; a directed cycle prevents one.

**WORK IT OUT**

1. Let $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$. Their intersection is $\{2, 3\}$; their union is $\{1, 2, 3, 4\}$.
2. In the pictured dependency graph, parse must finish before optimize, and optimize before emit.
3. With adjacency lists, a full BFS is $O(V + E)$. A dense matrix uses V squared entries even when few edges exist.

BFS work = $O(V + E)$; dense storage = $O(V^2)$

LESSON 12 / C++20

From idea to code.

Companion file: `examples/lesson_12.cpp`

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <set>
int main() {
    const std::set<int> a{1, 2, 3}, b{2, 3, 4};
    std::set<int> common;
    std::set_intersection(a.begin(), a.end(),
                          b.begin(), b.end(),
                          std::inserter(common, common.end()));
    for (int x : common) std::cout << x << ' ';
    std::cout << '\n';
}
```

EXPECTED OUTPUT

2 3

`set_intersection` requires sorted ranges with compatible ordering. `std::set` supplies that here. A hash-set iteration order would not meet this precondition.

WATCH THE BOUNDARY

An ordered container needs a strict weak ordering. Comparators based on changing state, inconsistent tolerances or unhandled NaNs can violate that contract.

TRY IT BEFORE READING THE ANSWER

A graph has one million vertices and three million edges. How many bytes would a one-bit-per-entry adjacency matrix require?

Answer 12 • page 85

LESSON 13 / UNDERSTAND

Count outcomes before using Bayes

Conditional probability changes which group you are counting.

Detectors, retries and reliability estimates become less confusing when probabilities are converted into counts.

$P(A|B)$ means the fraction of the B group that also belongs to A. It is not usually $P(B|A)$. The denominator is the group already known to apply.

For independent trials with failure probability p , the chance of no failures in n trials is $(1-p)^n$. Subtract from one to get at least one failure.

Independence is a model assumption. Failures sharing a network, power supply or overloaded service often occur together, so multiplying independent probabilities can be misleading.

100,000 runs

99

true alarms

999

false alarms

9.02%

alarms are faults

Sensitivity and the base rate answer different questions.

WORK IT OUT

1. Among 100,000 runs, suppose 100 contain faults. A detector finds 99 of them.
2. Of 99,900 healthy runs, a 1% false-alarm rate creates 999 false alarms.
3. There are 1,098 alarms in total. Only $99/1,098 = 9.02\%$ correspond to a fault. The base rate matters.

$$P(\text{fault} \mid \text{alarm}) = \text{true alarms} / \text{all alarms}$$

LESSON 13 / C11

From idea to code.

Companion file: `examples/lesson_13.c`

```
#include <math.h>
#include <stdio.h>
double any_failure(double p, unsigned n) {
    if (!isfinite(p) || p < 0 || p > 1) return NAN;
    if (n == 0 || p == 0) return 0;
    if (p == 1) return 1;
    return -expm1((double)n * log1p(-p));
}
int main(void) {
    printf("%.6f\n", any_failure(0.001, 100));
    printf("alarm means fault: %.2f%%\n", 100.0 * 99 / 1098);
    return 0;
}
```

EXPECTED OUTPUT

```
0.095208
alarm means fault: 9.02%
```

`log1p` and `expm1` preserve useful precision for small probabilities. This C lab assumes the usual target support for NaN; a production interface may prefer an explicit status result.

WATCH THE BOUNDARY

A 99% detection rate does not mean 99% of alerts are correct. Also, do not silently clamp an impossible probability into $[0,1]$; that hides bad input.

TRY IT BEFORE READING THE ANSWER

Two independent attempts each fail with probability 0.1. What is the probability that both fail? At least one fails?

Answer 13 • page 86

LESSON 14 / UNDERSTAND

Keep the shape of the data

One average cannot describe every user experience.

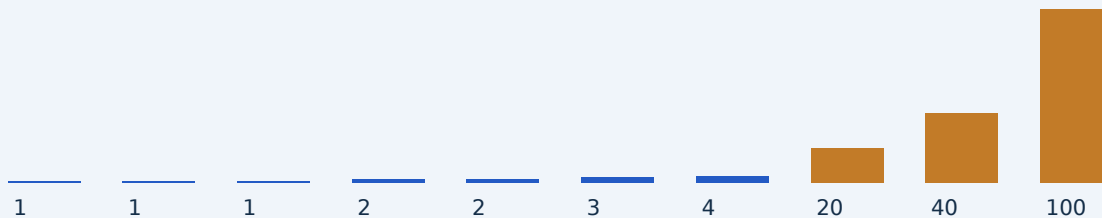
Latency and sensor data need summaries that match the question, especially when a few observations are extreme.

The mean measures total divided by count. The median describes the middle of sorted observations. Neither is automatically the right summary for every decision.

Variance measures squared spread. Population variance divides by n when describing the entire listed set; the usual unbiased sample-variance estimator divides by $n-1$ under independent, identically distributed sampling with finite variance.

A Bernoulli model describes one yes/no outcome. Binomial counts successes in independent equal-probability trials. Poisson models counts under a suitable constant-rate independent-increment process. A normal curve is not a universal law of latency.

Observed latency in milliseconds



WORK IT OUT

1. For $[1,1,1,2,2,3,4,20,40,100]$ ms, the total is 174 and the mean is 17.4 ms.
2. The middle two values are 2 and 3, so the median is 2.5 ms.
3. Using linear interpolation at position $q(n-1)$, p_{90} is 46 ms. Other percentile conventions can give different answers for this tiny sample.

mean = sum / n; sample variance = squared deviations / (n-1)

LESSON 14 / C++20

From idea to code.

Companion file: `examples/lesson_14.cpp`

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <stdexcept>
#include <vector>

double quantile(std::vector<double> x, double q) {
    if (x.empty() || !std::isfinite(q) || q < 0 || q > 1)
        throw std::domain_error("invalid quantile input");
    for (double v : x) if (!std::isfinite(v))
        throw std::domain_error("nonfinite sample");
    std::sort(x.begin(), x.end());
    const double p = q * static_cast<double>(x.size() - 1);
    const auto i = static_cast<std::size_t>(p);
    const auto j = std::min(i + 1, x.size() - 1);
    return std::lerp(x[i], x[j], p - static_cast<double>(i));
}

int main() {
    std::cout << quantile({1,1,1,2,2,3,4,20,40,100}, .9) << '\n';
}
```

EXPECTED OUTPUT

46

This teaching implementation targets ordinary in-memory datasets with size representable exactly as double. Project A adds mean, median and sample spread.

WATCH THE BOUNDARY

Do not average p99 values from different servers to obtain a global p99. Merge the underlying samples or compatible distribution summaries.

TRY IT BEFORE READING THE ANSWER

A hit costs 4 ns with probability .95 and a miss costs 80 ns with probability .05. What is the expected cost? Does it describe the tail?

[Answer 14](#) • [page 86](#)

LESSON 15 / UNDERSTAND

Measure uncertainty, not just speed

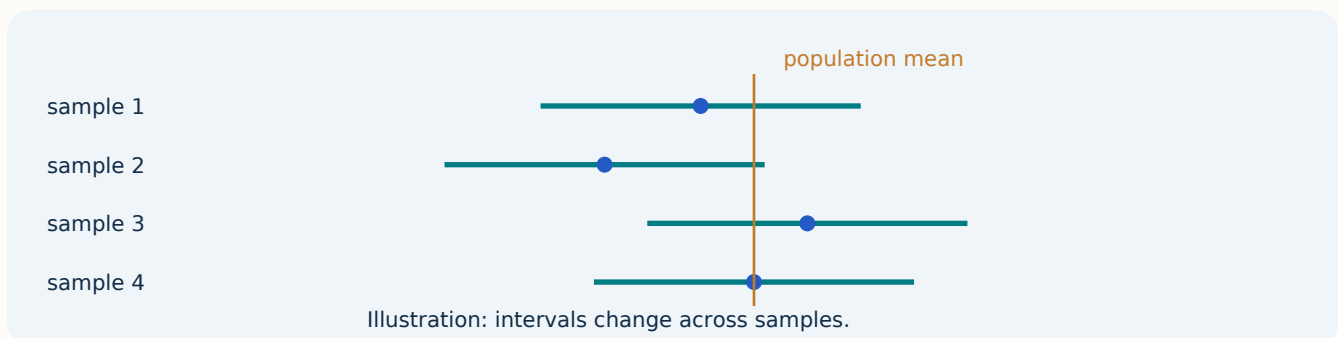
A measured speedup is an estimate from a noisy experiment.

Before celebrating a small improvement, ask whether the experiment can distinguish it from normal variation.

Independent observations with finite variance give a standard error of the mean estimated by $s/\sqrt{n}$. More observations help only when they add meaningful information.

For independent normal observations with unknown variance, a two-sided confidence interval for the mean uses a Student t multiplier. For large samples, 1.96 is a common approximate 95% multiplier under suitable conditions.

Confidence describes repeated coverage by the method. It does not mean that 95% of individual requests lie inside the interval. Correlated benchmark samples can make a naive interval too narrow.



WORK IT OUT

1. Suppose $n=100$ independent measurements have mean 50 ms and sample standard deviation 10 ms.
2. The estimated standard error is $10/\sqrt{100} = 1$ ms.
3. Using t about 1.984 for 99 degrees of freedom gives approximately [48.016, 51.984] ms for the mean, under the stated model.

$$SE = s / \sqrt{n}; \text{ interval} = \text{mean} \pm t * SE$$

LESSON 15 / C11

From idea to code.

Companion file: `examples/lesson_15.c`

```
#include <math.h>
#include <stdio.h>
int main(void) {
    const double mean_ms = 50.0;
    const double sample_sd_ms = 10.0;
    const double n = 100.0;
    const double t95 = 1.984; /* approximate, df = 99 */
    const double margin = t95 * sample_sd_ms / sqrt(n);
    printf("mean interval: [%.3f, %.3f] ms\n",
        mean_ms - margin, mean_ms + margin);
    return 0;
}
```

EXPECTED OUTPUT

mean interval: [48.016, 51.984] ms

This calculates an interval from supplied summary statistics; it is not a benchmark harness or a t-quantile implementation. The multiplier must change with the confidence level and sample size.

WATCH THE BOUNDARY

Use observable results, enough work per timed sample and a steady clock. Randomize or pair variant order, record compiler flags and hardware, and retain slow runs unless there is a justified exclusion rule.

TRY IT BEFORE READING THE ANSWER

How many independent observations are needed to halve the standard error if the underlying variance stays the same?

[Answer 15](#) • [page 86](#)

LESSON 16 / UNDERSTAND

Trade memory for a known error rate

Probabilistic data structures make a deliberate approximation.

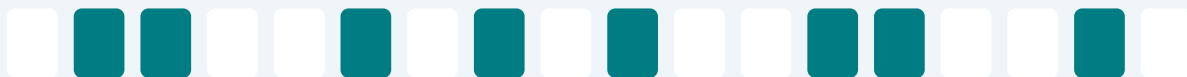
Bloom filters and randomized algorithms are useful when their assumptions and error costs are explicit.

A hash maps a large key space to fewer outputs, so collisions are unavoidable. Under approximately uniform independent hashes, collision probability grows roughly with the number of pairs, not just the number of keys.

A Bloom filter marks k bit positions per inserted key in m bits. An unset required bit proves absence; all set bits indicate possible presence. Correct insertion and lookup with consistent hashes give no false negatives in the basic, nondeleting design.

For n inserted keys, the common approximation is $(1 - \exp(-kn/m))^k$. Around $k = (m/n) \ln(2)$, the trade is efficient. The model assumes suitably distributed probes.

An unset queried bit proves absence.



All queried bits set means: check the original store.

WORK IT OUT

1. Give one million keys ten million bits: ten bits per key, or 1.25 decimal MB.
2. The approximate optimum probe count is $10 \ln(2) = 6.93$, so try $k=7$.
3. The approximate false-positive rate is 0.00819, or 0.819%. Confirm membership in the authoritative store.

$$p \sim (1 - \exp(-kn/m))^k$$

LESSON 16 / C++20

From idea to code.

Companion file: `examples/lesson_16.cpp`

```
#include <cmath>
#include <iomanip>
#include <iostream>
int main() {
    const double bits_per_key = 10.0;
    const int probes = 7;
    const double occupied = -std::expm1(-probes / bits_per_key);
    const double fp = std::pow(occupied, probes);
    std::cout << std::fixed << std::setprecision(3)
                << 100.0 * fp << "%\n";
}
```

EXPECTED OUTPUT

0.819%

This is a sizing model, not a filter implementation. Tracking the actual key count matters: inserting more keys fills more bits and increases false positives.

WATCH THE BOUNDARY

For simulation, use a suitable engine and distribution rather than `rng()%N` when that introduces modulo bias. C++ distribution output sequences need not match across library implementations, even with the same engine seed.

TRY IT BEFORE READING THE ANSWER

If every false positive costs a database lookup, approximately how many extra lookups occur among 100,000 queries for absent keys?

Answer 16 • page 86

LESSON 17 / UNDERSTAND

A vector is a list with geometry

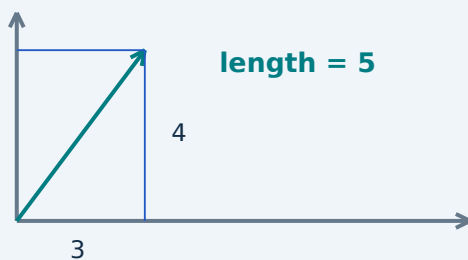
The dot product measures how strongly two directions agree.

The same operation supports lighting, similarity search, projections and machine-learning layers.

A vector is an ordered list of components. A geometric vector also has a coordinate frame and units. A feature vector needs a consistent feature order and scaling.

The dot product multiplies matching components and adds them. Its geometric form is $\text{length}(a) \times \text{length}(b) \times \cos(\text{angle})$. For unit vectors it becomes cosine similarity.

The Euclidean norm is the square root of the sum of squared components. For two or three components, hypot helps avoid unnecessary intermediate overflow and underflow. Normalizing a zero vector is undefined.



$$(3,4) / 5 = (0.6,0.8)$$

A unit vector keeps direction.

WORK IT OUT

1. For $a=(3,4)$, the length is 5. Dividing each component by 5 gives $(0.6,0.8)$.
2. Dotting this unit vector with $(1,0)$ gives 0.6, the cosine of their angle.
3. Projecting a onto the x-axis keeps $(3,0)$. The discarded component is perpendicular to that axis.

$$\mathbf{a} \text{ dot } \mathbf{b} = \text{sum}(\mathbf{a}[\mathbf{i}]\mathbf{b}[\mathbf{i}]); \text{ cosine} = (\mathbf{a} \text{ dot } \mathbf{b})/(\|\mathbf{a}\|\|\mathbf{b}\|)$$

LESSON 17 / C11

From idea to code.

Companion file: `examples/lesson_17.c`

```
#include <math.h>
#include <stdio.h>
int main(void) {
    const double x = 3.0, y = 4.0;
    const double length = hypot(x, y);
    if (!(length > 0) || !isfinite(length)) return 1;
    const double ux = x / length, uy = y / length;
    printf("length: %.1f\n", length);
    printf("unit: (%.1f, %.1f)\n", ux, uy);
    printf("cosine with x-axis: %.1f\n", ux);
    return 0;
}
```

EXPECTED OUTPUT

```
length: 5.0
unit: (0.6, 0.8)
cosine with x-axis: 0.6
```

This example has bounded finite inputs. High-dimensional vectors need a scaled norm or a vetted linear-algebra routine when magnitudes are extreme.

WATCH THE BOUNDARY

Two vectors from different coordinate frames cannot be compared meaningfully until their frames agree. For features, a large unit scale can dominate the dot product.

TRY IT BEFORE READING THE ANSWER

Compute $(1,2,3)$ dot $(4,0,-1)$. What is the cosine between $(1,0)$ and $(0,1)$?

Answer 17 • page 86

LESSON 18 / UNDERSTAND

Check matrix shapes before values

Every matrix multiplication is a collection of dot products.

Shape checking finds mistakes before an array index or a numerical value is examined.

A matrix with m rows and n columns can map an n -component vector to an m -component vector. Output row i is the dot product of row i with the input.

Multiplying A of shape m by k with B of shape k by n yields m by n . The inner dimensions match because each dot product has k terms.

Solving $Ax=b$ asks for an unknown vector. Do not form an explicit inverse just to solve a system. Use a suitable factorization: pivoted LU for general square systems, QR for many least-squares problems, and SVD when rank or conditioning needs special attention.

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}_{2 \times 2} \times \begin{bmatrix} 4 \\ 5 \end{bmatrix}_{2 \times 1} = \begin{bmatrix} 13 \\ 19 \end{bmatrix}_{2 \times 1}$$

WORK IT OUT

1. Let A have rows $[2,1]$ and $[1,3]$, and $x=[4,5]$.
2. The first output is $2 \times 4 + 1 \times 5 = 13$. The second is $1 \times 4 + 3 \times 5 = 19$.
3. Conversely, solving $Ax=[13,19]$ recovers $[4,5]$ here. Nearly dependent rows can make such recovery highly sensitive to input error.

$$A(m \times k) B(k \times n) \rightarrow C(m \times n)$$

LESSON 18 / C++20

From idea to code.

Companion file: `examples/lesson_18.cpp`

```
#include <array>
#include <iostream>
int main() {
    const std::array<std::array<double, 2>, 2> a{{
        {2.0, 1.0}, {1.0, 3.0}
    }};
    const std::array<double, 2> x{4.0, 5.0};
    std::array<double, 2> y{};
    for (std::size_t i = 0; i < 2; ++i)
        for (std::size_t j = 0; j < 2; ++j)
            y[i] += a[i][j] * x[j];
    std::cout << y[0] << ' ' << y[1] << '\n';
}
```

EXPECTED OUTPUT

13 19

The zero initialization of `y` is essential before accumulation. The fixed dimensions make every loop bound and shape visible.

WATCH THE BOUNDARY

Row-major storage and row-vector algebra are different choices. Storage order does not decide which side a vector multiplies on. Write the mathematical convention explicitly.

TRY IT BEFORE READING THE ANSWER

What shape results from a batch `X` of 32 rows and 8 features multiplied by weights `W` of 8 rows and 3 outputs?

Answer 18 • page 86

LESSON 19 / UNDERSTAND

Keep coordinates in their frames

A position only makes sense relative to a coordinate frame.

Graphics and robotics bugs often come from composing correct transforms in the wrong order.

A two-dimensional rotation through angle t maps (x,y) to $(\cos(t)x - \sin(t)y, \sin(t)x + \cos(t)y)$. Trigonometric library functions use radians.

Translation moves a point; it does not move a direction vector. Homogeneous coordinates encode this difference with a final component of 1 for points and 0 for directions.

With column vectors, the rightmost matrix acts first. A transform `T_world_from_local` names its source and destination, making composition easier to check.

Rotate, then translate

$(1,0) \longrightarrow (2,1)$

Translate, then rotate

$(1,0) \longrightarrow (0,3)$

The same two operations can produce different positions.

WORK IT OUT

1. Rotate point $(1,0)$ by 90 degrees counterclockwise. It becomes $(0,1)$, up to floating rounding.
2. Then translate by $(2,0)$. The result is $(2,1)$.
3. Reverse the order: translate to $(3,0)$, then rotate. The result is $(0,3)$. Order is part of the operation.

```
world_point = translation(rotation(local_point))
```

LESSON 19 / C++20

From idea to code.

Companion file: `examples/lesson_19.cpp`

```
#include <cmath>
#include <iostream>
#include <numbers>
struct P { double x, y; };
P rotate(P p, double radians) {
    const double c = std::cos(radians), s = std::sin(radians);
    return {c*p.x - s*p.y, s*p.x + c*p.y};
}
int main() {
    P p = rotate({1, 0}, std::numbers::pi / 2);
    p.x += 2;
    std::cout << p.x << ' ' << p.y << '\n';
}
```

EXPECTED OUTPUT

2 1

The exact mathematical cosine of $\pi/2$ is zero, but the floating result can be a tiny residual. Compare using the application's geometric tolerance.

WATCH THE BOUNDARY

Handedness, angle units and row/column conventions belong in the interface. A quaternion can represent 3D rotation compactly, but it still needs a convention and normalization policy.

TRY IT BEFORE READING THE ANSWER

Why does adding a translation to a velocity vector usually make no sense? What unit should an angular velocity have?

Answer 19 • page 87

LESSON 20 / UNDERSTAND

A derivative is local sensitivity

A derivative tells you how the output responds to a small input change.

Gradients guide optimization and also reveal when a result is sensitive to uncertain measurements.

For $y=f(x)$, the derivative is the limiting slope of nearby changes. Around x , a small change dx produces approximately $f'(x)dx$ in the output.

With several inputs, the gradient collects one partial derivative per input. Each component asks what happens when that input changes while the others stay fixed.

The chain rule connects stages: if y depends on u and u depends on x , then $dy/dx = (dy/du)(du/dx)$. Backpropagation repeatedly applies this rule while reusing intermediate values.

**WORK IT OUT**

1. Take $f(x)=(x-3)^2$. The derivative is $2(x-3)$.
2. At $x=5$ the slope is 4. A change of $+0.01$ predicts an output increase of about 0.04.
3. The exact increase is $4.0401-4 = 0.0401$. The small discrepancy is the curvature term.

local change $\approx$ slope $\times$ input change

LESSON 20 / C11

From idea to code.

Companion file: `examples/lesson_20.c`

```
#include <math.h>
#include <stdio.h>
double f(double x) { return (x - 3) * (x - 3); }
int main(void) {
    const double x = 5.0, h = 1e-5;
    const double numeric = (f(x+h) - f(x-h)) / (2*h);
    const double analytic = 2*(x-3);
    printf("analytic: %.6f\n", analytic);
    printf("numeric:  %.6f\n", numeric);
    return 0;
}
```

EXPECTED OUTPUT

```
analytic: 4.000000
numeric: 4.000000
```

A central difference is useful for checking a derivative at smooth points. Making h extremely small can increase roundoff error; compare several sensible step sizes.

WATCH THE BOUNDARY

A derivative is not an average slope over any arbitrary interval. At a nondifferentiable kink, a central-difference check may not match the chosen training subgradient.

TRY IT BEFORE READING THE ANSWER

If $y = \sin(2x)$, what is dy/dx ? If a measured radius has small error dr , how sensitive is circle area πr^2 ?

[Answer 20](#) • [page 87](#)

LESSON 21 / UNDERSTAND

Train a model by reducing a loss

Learning adjusts parameters to reduce a stated error measure.

A tiny regression model exposes the mathematics shared by much larger learning systems.

For a line $\hat{y}=w*x+b$, mean squared error averages squared residuals. A residual is prediction minus target. The loss tells us what the optimization is trying to improve.

The gradients are averages of $2*\text{residual}*x$ for w and $2*\text{residual}$ for b . Compute both from the same current parameters before updating them.

Gradient descent subtracts learning_rate times gradient. A rate that is too large can oscillate or diverge; a very small one wastes iterations. Scaling features changes this geometry.

**WORK IT OUT**

1. For the simpler loss $L(w)=(w-3)^2$, start $w=0$ and $\text{rate}=0.1$.
2. The gradient is -6 , so w becomes $0-0.1*(-6)=0.6$.
3. Each new error is 0.8 times the previous error. The loss keeps falling toward zero for this particular rate and quadratic.

```
parameter <- parameter - learning_rate x gradient
```

LESSON 21 / C++20

From idea to code.

Companion file: `examples/lesson_21.cpp`

```
#include <iomanip>
#include <iostream>
int main() {
    double w = 0;
    const double rate = 0.1;
    for (int step = 0; step < 100; ++step) {
        const double gradient = 2 * (w - 3);
        w -= rate * gradient;
    }
    std::cout << std::fixed << std::setprecision(6) << w << '\n';
}
```

EXPECTED OUTPUT

3.000000

The result rounds to six decimals; it is not mathematically exact after a finite number of steps. Project C extends this loop into line fitting with an independent gradient check.

WATCH THE BOUNDARY

Training loss alone cannot establish generalization. Keep validation data separate, choose a baseline, and check whether the features would actually be available at prediction time.

TRY IT BEFORE READING THE ANSWER

For this same quadratic, what happens at rate 0.5, rate 1 and rate greater than 1?

Answer 21 • page 87

LESSON 22 / UNDERSTAND

Accumulation turns rates into totals

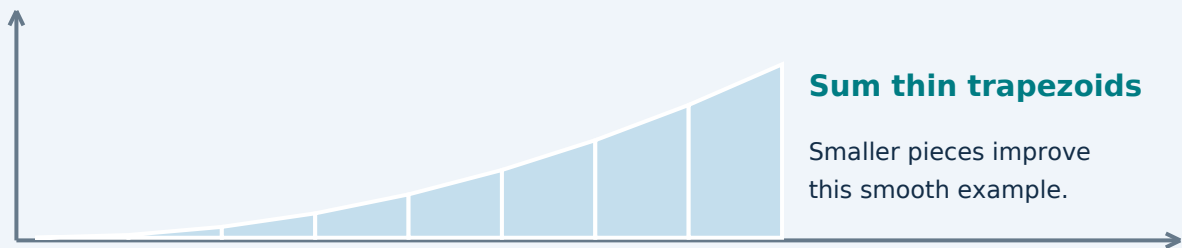
Integration adds many small contributions.

Area, energy and simulation all arise from adding a rate over a time or spatial interval.

If a rate is constant, total equals rate times interval. If it changes, split the interval into small pieces and approximate each contribution.

The trapezoid rule uses the average of the two endpoint heights on each piece. It integrates a straight line exactly in exact arithmetic.

An ordinary differential equation describes a state's rate of change. Explicit Euler uses the old state for every derivative. Updating velocity before position gives a different method, often called semi-implicit Euler in mechanics.



WORK IT OUT

1. Integrate $f(x)=x*x$ from 0 to 1 with two equal pieces of width 0.5.
2. The trapezoids have areas $0.5*(0+0.25)/2$ and $0.5*(0.25+1)/2$.
3. The total is 0.375; the exact integral is $1/3$. Finer pieces reduce the error for this smooth function.

$$\text{area per piece} \approx \text{width} \times (\text{left_height} + \text{right_height})/2$$

LESSON 22 / C11

From idea to code.

Companion file: `examples/lesson_22.c`

```
#include <stdio.h>
double f(double x) { return x*x; }
int main(void) {
    const int n = 1000;
    const double h = 1.0 / n;
    double area = 0;
    for (int i = 0; i < n; ++i) {
        const double left = i*h, right = (i+1)*h;
        area += h * (f(left) + f(right)) / 2;
    }
    printf("%.9f\n", area);
    return 0;
}
```

EXPECTED OUTPUT

0.333333500

The bounds and positive step count are fixed here. For a reusable integrator, validate the domain and inspect convergence under step refinement.

WATCH THE BOUNDARY

A smaller time step does not automatically make every method stable for every problem. For $x' = -kx$, explicit Euler decays only when $0 < k \cdot dt < 2$.

TRY IT BEFORE READING THE ANSWER

For $x' = -10x$, what interval of positive time steps makes explicit Euler decay? What happens at $dt = 0.2$?

[Answer 22](#) • [page 87](#)

LESSON 23 / UNDERSTAND

Signals are weighted neighborhoods

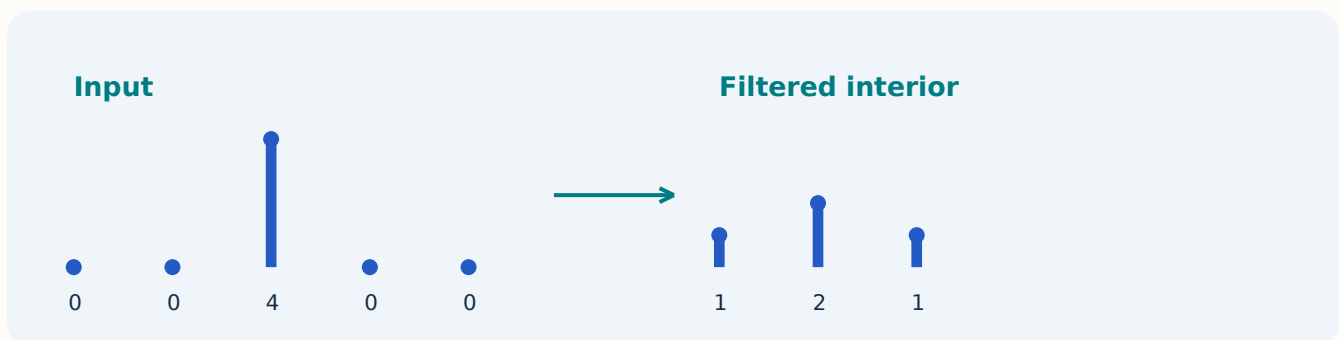
A filter combines nearby values with chosen weights.

The same idea smooths audio, blurs images and appears in convolutional neural networks.

A finite filter multiplies samples by weights and adds them. Weights summing to one preserve a constant signal away from boundary-policy effects.

Mathematical convolution reverses the kernel when sliding it. Many image and learning APIs implement cross-correlation without reversal. Symmetric kernels give the same result in either case.

Sampling loses information if higher frequencies fold into lower ones. A band-limited signal needs a sampling rate greater than twice its highest frequency for ideal reconstruction. Practical sampling requires an anti-alias filter and transition margin.

**WORK IT OUT**

1. Use weights [0.25,0.5,0.25] around samples [0,0,4,0,0].
2. The three valid interior outputs are 1,2,1.
3. An FFT computes a discrete Fourier transform efficiently. FFT multiplication naturally gives circular convolution; sufficient padding is needed for linear convolution.

$$y[i] = 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1]$$

LESSON 23 / C11

From idea to code.

Companion file: `examples/lesson_23.c`

```
#include <stdio.h>
int main(void) {
    const double x[5] = {0, 0, 4, 0, 0};
    for (size_t i = 1; i < 4; ++i) {
        const double y = .25*x[i-1] + .5*x[i] + .25*x[i+1];
        printf("%.1f%s", y, i == 3 ? "\n" : " ");
    }
    return 0;
}
```

EXPECTED OUTPUT

1.0 2.0 1.0

This lab emits only positions with a complete neighborhood. Project B demonstrates a separate, explicit boundary rule for a two-dimensional image.

WATCH THE BOUNDARY

A blur that is correct for linear-light values need not be correct for encoded sRGB numbers. The interpretation of the samples is part of the calculation.

TRY IT BEFORE READING THE ANSWER

For linear convolution of sequences of lengths 4 and 3, what is the output length? How much FFT padding is sufficient?

[Answer 23](#) • [page 87](#)

LESSON 24 / UNDERSTAND

Count bytes as carefully as operations

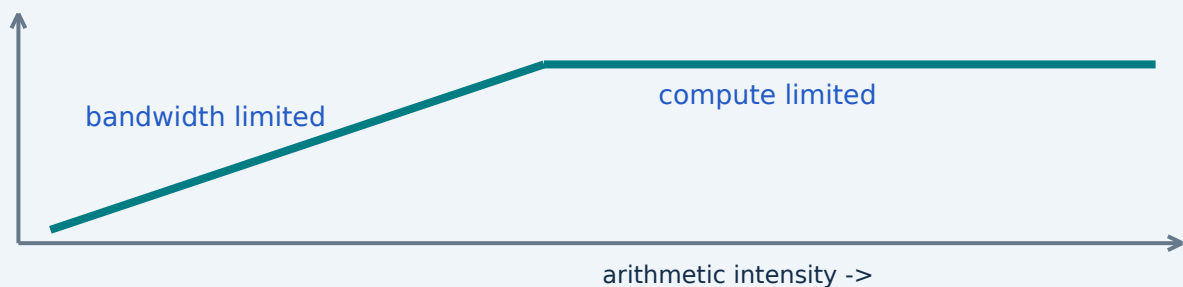
A fast processor cannot compute with data it has not received.

Bandwidth, locality and reuse often explain performance better than counting arithmetic alone.

Arithmetic intensity is useful operations per byte moved across a chosen memory boundary. State that boundary: register, cache and memory traffic are different counts.

Adding two arrays of doubles and writing one output performs one addition per 24 useful bytes per element. Real traffic can be larger because of cache behavior and write allocation.

A simple performance ceiling is the smaller of peak compute and bandwidth times arithmetic intensity. It is a model, not a promise. Blocking matrix multiplication improves reuse; SIMD increases arithmetic parallelism but cannot create bandwidth.

**WORK IT OUT**

1. A 256-bit vector can hold four 64-bit doubles. A 512-bit vector can hold eight.
2. At 50 billion useful bytes/s, a 24-byte-per-add stream has a bandwidth ceiling near 2.083 billion additions/s.
3. Unaligned access, tails, cache misses and reduction dependencies may lower measured throughput further.

$$\text{performance} \leq \min(\text{compute peak}, \text{bandwidth} \times \text{operations/byte})$$

LESSON 24 / C11

From idea to code.

Companion file: `examples/lesson_24.c`

```
#include <stdio.h>
int main(void) {
    const double bandwidth = 50e9;
    const double bytes_per_add = 3.0 * 8.0;
    const double ceiling = bandwidth / bytes_per_add;
    printf("ideal ceiling: %.3f Gadd/s\n", ceiling / 1e9);
    const size_t row = 7, col = 3, columns = 20;
    printf("row-major offset: %zu\n", row*columns + col);
    return 0;
}
```

EXPECTED OUTPUT

```
ideal ceiling: 2.083 Gadd/s
row-major offset: 143
```

The offset is in elements, not bytes. Dynamic indexing needs row and column bounds plus a checked allocation size. A contiguous `std::vector` matrix is not the same layout as nested vectors.

WATCH THE BOUNDARY

Floating-point reduction order changes under vectorization and parallelism. Fast-math options can discard assumptions required by compensated sums and finite-value checks.

TRY IT BEFORE READING THE ANSWER

What is the ideal minimum scan time for 5 GiB at 50 GiB/s? Is a 64-byte cache line guaranteed by C++?

Answer 24 • page 87

LESSON 25 / UNDERSTAND

Parallel work still has a serial part

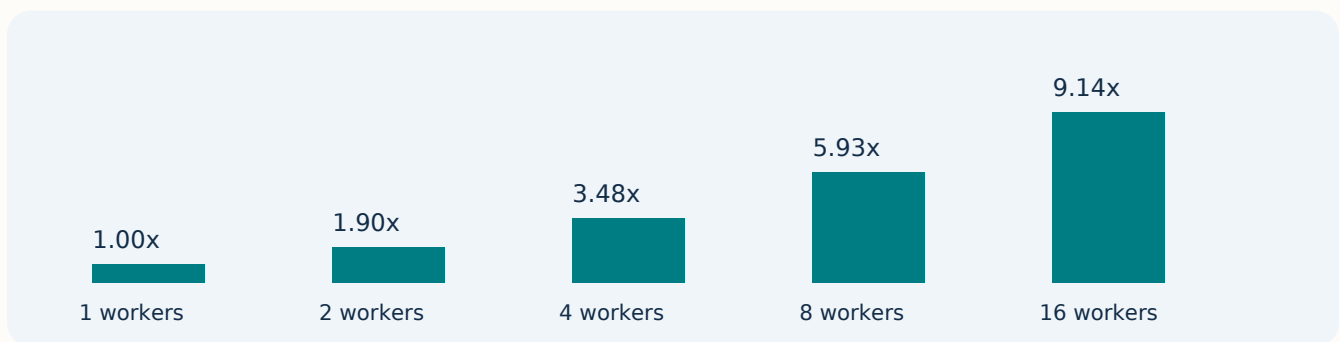
More workers help only where work can actually overlap.

Servers and parallel kernels need both a workload model and correct synchronization.

Amdahl's fixed-workload model splits execution into serial fraction s and perfectly parallel fraction $1-s$. With N workers, ideal speedup is $1/(s+(1-s)/N)$.

Little's law relates long-run average items in a stable system to its average arrival rate and time in that system: $L = \lambda * W$. Use one consistent system boundary and units.

Correct concurrency is about ordering and lifetime, not just arithmetic. A C++ data race has undefined behavior. Independent variables can still share a cache line and contend through false sharing.



WORK IT OUT

1. If 5% of a fixed workload is serial, the infinite-worker speedup ceiling is 20.
2. With 16 workers, ideal speedup is $1/(0.05+0.95/16) = 9.143$.
3. At 1,000 requests/s and a mean 20 ms in the measured system, Little's law gives 20 requests present on average.

$$S = 1/(s + (1-s)/N); L = \text{arrival_rate} \times \text{mean_time}$$

LESSON 25 / C++20

From idea to code.

Companion file: `examples/lesson_25.cpp`

```
#include <iomanip>
#include <iostream>
int main() {
    const double serial = 0.05, workers = 16;
    const double speedup = 1 / (serial + (1-serial)/workers);
    const double arrival_rate = 1000, mean_seconds = 0.02;
    std::cout << std::fixed << std::setprecision(3)
              << "ideal speedup: " << speedup << '\n';
    std::cout << "mean in system: "
              << arrival_rate * mean_seconds << '\n';
}
```

EXPECTED OUTPUT

```
ideal speedup: 9.143
mean in system: 20.000
```

This models a fixed workload and ignores parallel overhead. A scaled workload can behave differently. Little's law gives an average, not a queue capacity or percentile guarantee.

WATCH THE BOUNDARY

Atomic variables do not make an entire algorithm correct. Publication, memory ordering, reclamation and compound invariants still require a complete argument.

TRY IT BEFORE READING THE ANSWER

If $s=.10$ and $N=8$, what is ideal speedup? Why is average utilization below 100% not enough to prove a real-time schedule meets every deadline?

Answer 25 • page 88

LESSON 26 / UNDERSTAND

Prepare data without leaking the answer

A transformation must learn only from data available at training time.

Good modeling starts before the first gradient: with units, splits, missing values and a reproducible preprocessing contract.

Standardization subtracts a training mean and divides by a training scale. This can keep a large-unit feature from dominating distances and gradients.

Fit preprocessing on the training partition only. Apply the same stored parameters to validation, test and future inputs. Splitting after fitting all data leaks information.

A zero-variance feature needs a policy, such as producing zero and recording that it was constant. Random splitting is often wrong for time-ordered data or repeated observations from the same subject.

**WORK IT OUT**

1. Training values [10,20,30] have mean 20.
2. Using population scale for these three training values gives $\text{sqrt}((100+0+100)/3) = 8.164966$.
3. A future value 40 becomes $(40-20)/8.164966 = 2.449490$. It is allowed to lie outside the training range.

$$z = (x - \text{training_mean}) / \text{training_scale}$$

LESSON 26 / C++20

From idea to code.

Companion file: `examples/lesson_26.cpp`

```
#include <cmath>
#include <iomanip>
#include <iostream>
int main() {
    const double train[] = {10, 20, 30};
    double mean = 0;
    for (double x : train) mean += x / 3.0;
    double variance = 0;
    for (double x : train) variance += (x-mean)*(x-mean)/3.0;
    const double scale = std::sqrt(variance);
    if (!(scale > 0)) return 1;
    const double future = 40;
    std::cout << std::fixed << std::setprecision(6)
              << (future-mean)/scale << '\n';
}
```

EXPECTED OUTPUT

2.449490

The scale convention here is descriptive population standard deviation on the training set. The data are tiny and bounded; large streams need a stable online or two-pass implementation.

WATCH THE BOUNDARY

High test accuracy can coexist with leakage or a useless class imbalance. Compare with a simple baseline and select metrics that reflect the actual decision.

TRY IT BEFORE READING THE ANSWER

If training features are standardized separately from validation features using each partition's own mean, why is that a problem?

Answer 26 • page 88

LESSON 27 / UNDERSTAND

Turn scores into a stable loss

Equivalent formulas can have very different numerical behavior.

Classification relies on exponentials and logarithms, where a good algebraic form prevents avoidable overflow.

Softmax turns finite scores into positive weights summing to one in exact arithmetic. Subtracting the largest score before exponentiating keeps every exponential at most one.

For a target class t , cross-entropy is $-\log(p[t])$. Compute it from shifted scores as $\log(\sum(\exp(z[j]-m))) - (z[t]-m)$, rather than taking log of a probability that may round to zero.

For binary classification, a stable sigmoid uses $1/(1+\exp(-z))$ when z is nonnegative and $\exp(z)/(1+\exp(z))$ otherwise. Probabilities from a model still need calibration checks.

Stable probabilities for [1000,1001,1002]

0.0900



class 0

0.2447



class 1

0.6652



class 2

WORK IT OUT

1. Raw scores [1000,1001,1002] would overflow a direct binary64 exp calculation.
2. Subtract 1002 to get [-2,-1,0]. Their exponentials are approximately [0.1353,0.3679,1].
3. Normalize to [0.0900,0.2447,0.6652]. For the last class, loss is about 0.407606.

$$p[i] = \exp(z[i]-m) / \sum(\exp(z[j]-m))$$

LESSON 27 / C++20

From idea to code.

Companion file: `examples/lesson_27.cpp`

```
#include <algorithm>
#include <array>
#include <cmath>
#include <iomanip>
#include <iostream>
int main() {
    const std::array<double,3> z{1000, 1001, 1002};
    const double m = *std::max_element(z.begin(), z.end());
    std::array<double,3> e{};
    double sum = 0;
    for (std::size_t i=0; i<z.size(); ++i) {
        e[i] = std::exp(z[i]-m);
        sum += e[i];
    }
    std::cout << std::fixed << std::setprecision(6);
    for (double v : e) std::cout << v/sum << ' ';
    std::cout << "\nloss: " << std::log(sum)-(z[2]-m) << '\n';
}
```

EXPECTED OUTPUT

```
0.090031 0.244728 0.665241
loss: 0.407606
```

For these bounded finite inputs, max shifting avoids overflow. A general interface must define empty input, nonfinite logits and an out-of-range target; extreme differences can still underflow to zero.

WATCH THE BOUNDARY

Softmax weights are not automatically reliable real-world probabilities. A model may be confidently wrong on inputs unlike its training data.

TRY IT BEFORE READING THE ANSWER

What happens to the output if 50 is added to every score? What happens if all scores are multiplied by 10?

Answer 27 • page 88

LESSON 28 / UNDERSTAND

Backpropagation is the chain rule reused

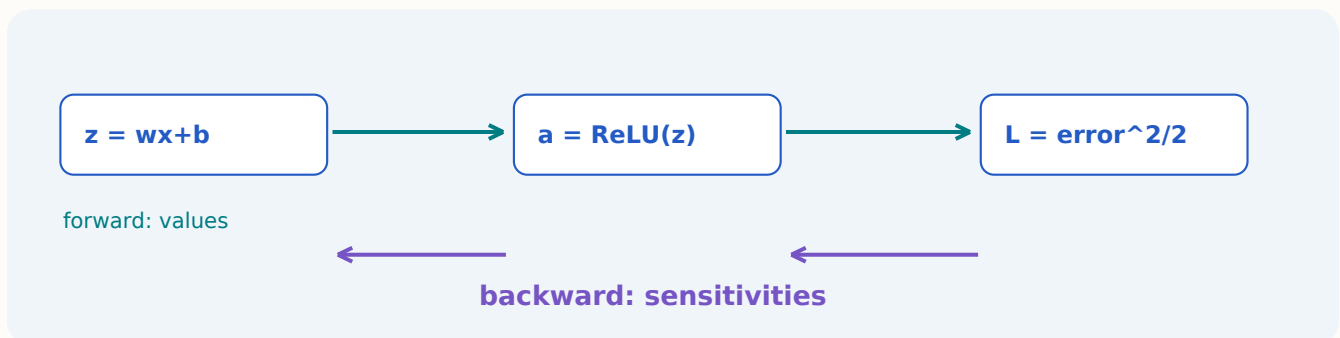
Follow the forward calculation, then send sensitivities backward.

One neuron is enough to understand why automatic differentiation works.

Let $z = w \cdot x + b$, $a = \max(0, z)$, and $L = (a - y)^2 / 2$. The half makes the first derivative of the loss simply $a - y$.

When $z > 0$, the ReLU derivative is 1. When $z < 0$, it is 0. At zero, the derivative does not exist; training implementations choose a convention, often zero.

Multiply local derivatives: $dL/dw = (a - y) \cdot \text{ReLU}'(z) \cdot x$ and $dL/db = (a - y) \cdot \text{ReLU}'(z)$. Reverse-mode differentiation shares these intermediate sensitivities across many parameters.



WORK IT OUT

1. Take $x=2$, $w=0.5$, $b=0$, $y=3$. The forward pass gives $z=1$, $a=1$ and $L=2$.
2. The output sensitivity is $a - y = -2$. The positive ReLU passes it through unchanged.
3. The weight gradient is -4 and the bias gradient is -2 . At rate 0.1 , new values are $w=0.9$ and $b=0.2$.

$$dL/dw = dL/da \times da/dz \times dz/dw$$

LESSON 28 / C++20

From idea to code.

Companion file: `examples/lesson_28.cpp`

```
#include <algorithm>
#include <iostream>
int main() {
    const double x=2, target=3, rate=0.1;
    double w=0.5, b=0;
    const double z=w*x+b;
    const double a=std::max(0.0, z);
    const double dz=(a-target)*(z>0 ? 1.0 : 0.0);
    const double dw=dz*x, db=dz;
    w -= rate*dw;
    b -= rate*db;
    const double next=std::max(0.0, w*x+b);
    std::cout << "w: " << w << " b: " << b << '\n';
    std::cout << "new loss: " << .5*(next-target)*(next-target) << '\n';
}
```

EXPECTED OUTPUT

```
w: 0.9 b: 0.2
new loss: 0.5
```

Both gradients come from the same forward pass. The new output is 2 and the loss is 0.5. One improving step is a local check, not proof that training will always succeed.

WATCH THE BOUNDARY

Updating w before computing a gradient that depends on the old w mixes parameter states. Keep forward values and gradients consistent, especially across a batch.

TRY IT BEFORE READING THE ANSWER

If z is strictly negative in this example, what are the gradients through ReLU? Why avoid checking a derivative exactly at $z=0$?

Answer 28 • [page 88](#)

LESSON 29 / UNDERSTAND

Similarity and attention share a dot product

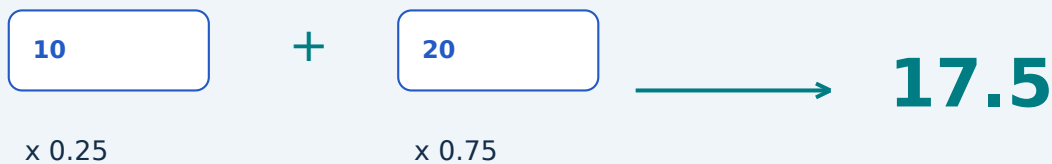
Attention is a data-dependent weighted average.

Embeddings and attention become concrete when their vector operations are separated from the model that learned them.

An embedding is a vector representation learned or designed for a purpose. Cosine similarity compares directions after normalization; a raw dot product also depends on magnitude.

Scaled dot-product attention forms scores $QK^T/\text{sqrt}(d)$, applies softmax across keys for each query, then combines the value vectors V with those weights.

With independent zero-mean unit-variance components, the dot-product variance grows with dimension d . Dividing by $\text{sqrt}(d)$ controls that scale under this motivating model. It is not a universal distribution guarantee.



A convex weighted average stays inside the value range.

WORK IT OUT

1. One query produces scores $[0, \ln(3)]$ for two keys. Exponentiation gives $[1, 3]$.
2. Softmax weights are $[0.25, 0.75]$. If the corresponding scalar values are 10 and 20, the output is 17.5.
3. For many queries and keys, standard dense attention creates a query-by-key score matrix; its size can grow quadratically with sequence length.

$$\text{attention}(Q, K, V) = \text{softmax}(QK^T / \text{sqrt}(d)) V$$

LESSON 29 / C++20

From idea to code.

Companion file: `examples/lesson_29.cpp`

```
#include <cmath>
#include <iostream>
int main() {
    const double s0=0, s1=std::log(3.0);
    const double m=s1;
    const double e0=std::exp(s0-m), e1=std::exp(s1-m);
    const double w0=e0/(e0+e1), w1=e1/(e0+e1);
    const double output=w0*10+w1*20;
    std::cout << w0 << ' ' << w1 << '\n';
    std::cout << output << '\n';
}
```

EXPECTED OUTPUT

```
0.25 0.75
17.5
```

This lab isolates the weighting step. Project D provides a complete tiny retrieval example with hand-written vectors; it does not pretend to train semantic embeddings.

WATCH THE BOUNDARY

A causal mask removes access to future positions. If every key is masked, ordinary softmax has no valid distribution; the implementation needs an explicit policy.

TRY IT BEFORE READING THE ANSWER

If Q has shape 5×4 , K is 7×4 and V is 7×3 , what are the score and output shapes?

Answer 29 • page 88

LESSON 30 / UNDERSTAND

Make quantization an error-budget decision

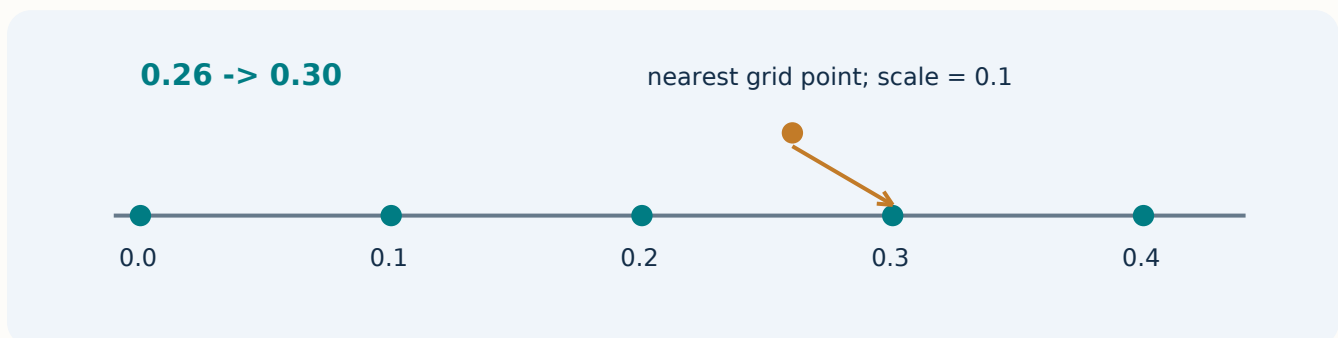
Fewer bits buy efficiency by accepting a measured approximation.

Integer inference and compact data formats need explicit scale, clipping, rounding and accumulator rules.

An affine quantizer maps a real value to $q = \text{round}(x/s) + z$, then clips q to a stored integer range. Dequantization returns $s \cdot (q - z)$. The scale s must be positive.

Without clipping, nearest rounding gives absolute error at most $s/2$ in exact arithmetic. Clipping can produce much larger error outside the represented interval.

Integer products usually accumulate in a wider type. For symmetric magnitudes bounded by 127, each product is at most 16,129 in magnitude; the dot-product length determines whether the sum fits. Bias and zero-point terms also matter.



WORK IT OUT

1. Take scale $s=0.1$ and zero point $z=0$, using stored integers from -127 to 127.
2. For $x=0.26$, nearest rounding gives $q=3$. Reconstruction is 0.3, so error is 0.04.
3. For $x=20$, clipping gives $q=127$ and reconstruction 12.7. The 7.3 error is clipping error, far larger than $s/2$.

$$\hat{x} = \text{scale} \times (\text{quantized} - \text{zero_point})$$

LESSON 30 / C++20

From idea to code.

Companion file: `examples/lesson_30.cpp`

```
#include <algorithm>
#include <cmath>
#include <iostream>
int main() {
    const double x=0.26, scale=0.1;
    const double rounded=std::round(x/scale);
    const double bounded=std::clamp(rounded, -127.0, 127.0);
    const int q=static_cast<int>(bounded);
    const double restored=scale*q;
    std::cout << "q: " << q << " restored: " << restored << '\n';
    std::cout << "error: " << std::abs(restored-x) << '\n';
}
```

EXPECTED OUTPUT

q: 3 restored: 0.3
error: 0.04

The finite bounded inputs make the cast safe. `std::round` uses halfway-away-from-zero; many inference formats use another policy. Match the target format instead of assuming they agree.

WATCH THE BOUNDARY

Choose calibration data that represent deployment inputs, then measure end-task quality as well as numerical error. Smaller tensors do not guarantee lower latency on every backend.

TRY IT BEFORE READING THE ANSWER

With scale 0.02 and no clipping, what is the nearest-rounding error bound? Does `int8` storage justify accumulating a long dot product in `int8`?

[Answer 30](#) • [page 88](#)

PROJECT A

A latency report that tells the truth

Combine summaries, percentiles and numerical care.

You receive ten request latencies in milliseconds. The first task is to describe this particular sample, not to declare the performance of every future request.

The program computes mean and sample standard deviation in one pass with Welford's update, then sorts a copy for interpolated percentiles. The large tail makes the difference between typical latency and total cost visible.

Welford keeps a running mean and the sum $M2$ of squared deviations from that evolving mean. Each new observation first changes the mean, then contributes $\text{delta} * (x - \text{new_mean})$ to $M2$. This avoids subtracting two huge, nearly equal sums.

DECISIONS BEFORE IMPLEMENTATION

Input contract. A fixed, nonempty list of finite, nonnegative latencies. The example is small; external data must be validated before aggregation.

Percentile rule. Use position $q * (n - 1)$ in the sorted list and interpolate between adjacent values. State this rule when comparing tools.

Variance rule. Use $M2 / (n - 1)$ for the sample variance. It requires at least two samples; there is no sample spread estimate from one observation.

Interpretation. Describe the observations. Do not manufacture a confidence interval from ten strongly skewed values without an appropriate model.

CONNECTIONS

Use the lesson tools deliberately: units and ranges first, a hand-computed input second, then a complete program and checks that should remain true after changes.

project_a_latency.cpp

The complete program.

```

#include <algorithm>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>
int main() {
    std::vector<double> x{1,1,1,2,2,3,4,20,40,100};
    std::size_t n=0;
    double mean=0, m2=0;
    for (double v : x) {
        if (!std::isfinite(v) || v<0) return 1;
        ++n;
        const double delta=v-mean;
        mean += delta/static_cast<double>(n);
        m2 += delta*(v-mean);
    }
    if (n<2) return 1;
    std::sort(x.begin(), x.end());
    const auto quantile = [&](double q) {
        const double p=q*static_cast<double>(n-1);
        const auto i=static_cast<std::size_t>(p);
        const auto j=std::min(i+1, n-1);
        return std::lerp(x[i], x[j], p-static_cast<double>(i));
    };
    std::cout << std::fixed << std::setprecision(3);
    std::cout << "n: " << n << '\n';
    std::cout << "mean: " << mean << " ms\n";
    std::cout << "sample sd: "
        << std::sqrt(m2/static_cast<double>(n-1)) << " ms\n";
    std::cout << "median: " << quantile(.5) << " ms\n";
    std::cout << "p90: " << quantile(.9) << " ms\n";
    std::cout << "max: " << x.back() << " ms\n";
}

```

Build and run this file as its own program. The companion CMake project provides a separate executable target with the same base name.

Explain what the output means.

EXPECTED OUTPUT

n: 10
mean: 17.400 ms
sample sd: 31.638 ms
median: 2.500 ms
p90: 46.000 ms
max: 100.000 ms

The mean is 17.4 ms even though seven of the ten values are at most 4 ms. The 100 ms observation contributes more than half the total time.

The median is a better description of the middle request here. The maximum identifies the worst observed request; it is not a bound on future latency.

The p90 result is an interpolation convention, not an observation that necessarily occurred. With only ten samples, tail estimates are extremely coarse.

CHECKS THAT MEAN SOMETHING

1. Replace the list by ten identical 5 ms values: mean and every percentile must be 5, and standard deviation must be zero.
2. Add 10 to every sample: the mean and percentiles shift by 10, while standard deviation stays unchanged within rounding.
3. Multiply all observations by two: all reported time-valued summaries must double.

TAKE IT FURTHER

Next, load measurements from your own benchmark. Keep raw units, count, environment and collection method with the report. For very large streams, use a documented quantile sketch and record its error contract.

PROJECT B

A small image filter in plain C

Turn a local weighted average into a visible transformation.

An image can be treated as a two-dimensional field of numbers. This project blurs a synthetic 5x5 grayscale image with a 3x3 box filter.

The input has a single bright center value of 9. Each output averages the available neighbors. At the image border, this implementation uses only valid neighbors and divides by their actual count.

The program writes into a separate output array. Updating the input in place would let later pixels read already-blurred values, changing the mathematical operation.

DECISIONS BEFORE IMPLEMENTATION

Input meaning. Small linear-intensity grayscale values. These are not encoded sRGB bytes, so no transfer-function conversion is needed.

Kernel. Equal weights across the 3x3 neighborhood. In the interior each weight is 1/9.

Boundary rule. Crop the neighborhood at the border and renormalize. This is a spatially varying rule near borders, not the same as zero-padding.

Storage. Two fixed 5x5 arrays make bounds explicit. Dynamic images need checked dimensions and allocation sizes.

CONNECTIONS

Use the lesson tools deliberately: units and ranges first, a hand-computed input second, then a complete program and checks that should remain true after changes.

project_b_blur.c

The complete program.

```
#include <stdio.h>
int main(void) {
    enum { H=5, W=5 };
    const double input[H][W] = {
        {0,0,0,0,0}, {0,0,0,0,0}, {0,0,9,0,0},
        {0,0,0,0,0}, {0,0,0,0,0}
    };
    double output[H][W] = {{0}};
    for (int r=0; r<H; ++r) {
        for (int c=0; c<W; ++c) {
            double sum=0;
            int count=0;
            for (int dr=-1; dr<=1; ++dr) {
                for (int dc=-1; dc<=1; ++dc) {
                    const int rr=r+dr, cc=c+dc;
                    if (rr>=0 && rr<H && cc>=0 && cc<W) {
                        sum += input[rr][cc];
                        ++count;
                    }
                }
            }
            output[r][c]=sum/count;
        }
    }
    for (int r=0; r<H; ++r) {
        for (int c=0; c<W; ++c)
            printf("%.1f%s", output[r][c], c==W-1 ? "\n" : " ");
    }
    return 0;
}
```

Build and run this file as its own program. The companion CMake project provides a separate executable target with the same base name.

Explain what the output means.

EXPECTED OUTPUT

```
0.0 0.0 0.0 0.0 0.0
0.0 1.0 1.0 1.0 0.0
0.0 1.0 1.0 1.0 0.0
0.0 1.0 1.0 1.0 0.0
0.0 0.0 0.0 0.0 0.0
```

The center value is shared among the nine interior positions whose neighborhoods contain it. In this example each becomes 1.

A constant input remains constant, including at the border, because each output divides by its actual number of neighbors.

Total brightness happens to be preserved for this centered impulse. This boundary rule does not preserve the global sum for every possible image; moving the impulse near an edge reveals the distinction.

CHECKS THAT MEAN SOMETHING

1. Replace every pixel with 7: every output must be 7.
2. Move the impulse to a corner and compute the four nonzero outputs by hand: $9/4$, $9/6$, $9/6$ and $9/9$.
3. Check that changing one input affects only output positions whose neighborhoods include it.

TAKE IT FURTHER

Next, compare zero-padding, reflection and clamped borders. They are different mathematical definitions. For a production image, define color space, channel layout and rounding before optimizing.

PROJECT C

Fit a line and check its gradient

Build a complete, deterministic learning loop.

The training pairs are $x=[-2,-1,0,1,2]$ and $y=[-3,-1,1,3,5]$. They follow $y=2x+1$. We know the answer so we can audit the learning process.

The model predicts $w*x+b$. Its loss is the mean squared residual. We compute gradients from every training point, average them, then update both parameters together.

Before training, a central finite difference checks the weight derivative at $w=0, b=0$. It should agree with the analytic gradient, approximately -8. This test catches missing factors and sign errors.

DECISIONS BEFORE IMPLEMENTATION

Data design. Centered inputs make the arithmetic easy to inspect. Real data need a deliberate split and training-only preprocessing.

Loss. Mean squared error, without a one-half factor. Therefore both derivatives contain a factor of 2.

Learning rate. 0.1 is suitable for this bounded quadratic. It is not a transferable default for arbitrary features.

Success criterion. Parameters approach $w=2$ and $b=1$; a held-out point $x=3$ predicts 7. These synthetic results demonstrate mechanics, not real-world model quality.

CONNECTIONS

Use the lesson tools deliberately: units and ranges first, a hand-computed input second, then a complete program and checks that should remain true after changes.

project_c_regression.cpp

The complete program.

```

#include <array>
#include <iomanip>
#include <iostream>
const std::array<double,5> xs{-2,-1,0,1,2};
const std::array<double,5> ys{-3,-1,1,3,5};
double loss(double w, double b) {
    double total=0;
    for (std::size_t i=0; i<xs.size(); ++i) {
        const double r=w*xs[i]+b-ys[i];
        total += r*r;
    }
    return total/static_cast<double>(xs.size());
}
int main() {
    const double h=1e-5;
    const double numeric=(loss(h,0)-loss(-h,0))/(2*h);
    double w=0, b=0;
    for (int step=0; step<200; ++step) {
        double dw=0, db=0;
        for (std::size_t i=0; i<xs.size(); ++i) {
            const double r=w*xs[i]+b-ys[i];
            dw += 2*r*xs[i];
            db += 2*r;
        }
        const double n=static_cast<double>(xs.size());
        w -= 0.1*dw/n;
        b -= 0.1*db/n;
    }
    std::cout << std::fixed << std::setprecision(6);
    std::cout << "numeric dw at start: " << numeric << '\n';
    std::cout << "w: " << w << " b: " << b << '\n';
    std::cout << "training loss: " << loss(w,b) << '\n';
    std::cout << "prediction at 3: " << 3*w+b << '\n';
}

```

Build and run this file as its own program. The companion CMake project provides a separate executable target with the same base name.

Explain what the output means.

EXPECTED OUTPUT

```
numeric dw at start: -8.000000  
w: 2.000000 b: 1.000000  
training loss: 0.000000  
prediction at 3: 7.000000
```

At the initial parameters, the residuals are $-y$. Summing $2 \cdot \text{residual} \cdot x$ and dividing by five gives -8 ; the finite difference confirms this locally.

The bias gradient starts at -2 . Both parameters move toward the known line. The printed zero loss is rounded, not a claim of exact arithmetic.

If inputs are multiplied by a thousand without adjusting the rate, the weight gradient can become much larger. Feature scale changes optimization behavior.

CHECKS THAT MEAN SOMETHING

1. Check both derivatives at a second point such as $w=.7, b=-.2$, not just at zero.
2. Add a constant c to every target: the fitted bias should increase by c while the slope remains about 2.
3. Compare predictions against a constant-mean baseline and evaluate on data not used for fitting.

TAKE IT FURTHER

Next, add controlled noise and hold out a test set. Record training and test losses separately. For larger or ill-conditioned least squares, use QR or SVD rather than explicitly inverting $X\text{-transpose-X}$.

PROJECT D

Retrieve by direction, then explain the weights

Inspect the arithmetic behind a miniature vector search.

Three items have illustrative two-dimensional vectors. The query is $[1,1]$. We normalize the query and each item, compute cosine scores, and rank the items.

The vectors are hand-written teaching data. The program does not generate semantic embeddings, understand language or validate a real retrieval model.

We also form stable softmax weights at temperature 0.5. Ranking uses cosine scores; the weights illustrate how the same scores can mix associated values.

DECISIONS BEFORE IMPLEMENTATION

Vectors. $A=[1,0]$, $B=[1,1]$, $C=[0,1]$. B aligns exactly with the query; A and C tie.

Normalization. Reject zero-length vectors. All components here are small and finite.

Ranking. Higher cosine is better. Equal scores are ordered by item ID for deterministic presentation.

Weight meaning. These normalized weights are not probabilities that an item is correct. Temperature controls concentration and must have a positive value.

CONNECTIONS

Use the lesson tools deliberately: units and ranges first, a hand-computed input second, then a complete program and checks that should remain true after changes.

project_d_retrieval.cpp

The complete program.

```

#include <algorithm>
#include <array>
#include <cmath>
#include <iomanip>
#include <iostream>
struct Item { char id; double x, y, score, weight; };
int main() {
    std::array<Item,3> items{{
        {'A',1,0,0,0}, {'B',1,1,0,0}, {'C',0,1,0,0}
    }};
    const double qx=1/std::sqrt(2.0), qy=qx;
    const double temperature=0.5;
    double max_score=-1;
    for (auto& item : items) {
        const double norm=std::hypot(item.x,item.y);
        if (!(norm>0) || !std::isfinite(norm)) return 1;
        item.score=qx*(item.x/norm)+qy*(item.y/norm);
        max_score=std::max(max_score,item.score);
    }
    double sum=0;
    for (auto& item : items) {
        item.weight=std::exp((item.score-max_score)/temperature);
        sum += item.weight;
    }
    for (auto& item : items) item.weight /= sum;
    std::sort(items.begin(),items.end(),[](const Item& a,const Item& b){
        if (a.score!=b.score) return a.score>b.score;
        return a.id<b.id;
    });
    std::cout << std::fixed << std::setprecision(6);
    for (const auto& item : items)
        std::cout << item.id << " cosine=" << item.score
            << " weight=" << item.weight << '\n';
}

```

Build and run this file as its own program. The companion CMake project provides a separate executable target with the same base name.

Explain what the output means.

EXPECTED OUTPUT

```
B cosine=1.000000 weight=0.473186
A cosine=0.707107 weight=0.263407
C cosine=0.707107 weight=0.263407
```

B is the closest direction. The equal scores of A and C are resolved by the explicit ID rule, not by an accidental container order.

Changing temperature changes weights but not ranking for a positive temperature. Multiplying an item vector by a positive constant leaves cosine unchanged.

If there are many stored vectors, exact scanning costs $O(\text{number_of_items} \times \text{dimension})$. Approximate indexes trade retrieval recall against memory and latency; test that trade on representative queries.

CHECKS THAT MEAN SOMETHING

1. Use query [1,0]: A should rank first.
2. Multiply B by ten: cosine scores should remain unchanged within rounding.
3. Check every weight is nonnegative and the total is close to one. Reject a zero query before normalization.

TAKE IT FURTHER

Next, replace the teaching vectors with real embeddings from a defined model and evaluate against relevance judgments. Keep model version, normalization and distance metric consistent across stored items and queries.

KEEP GOING / SPECIALTY GUIDE

Graphics and geometry

Keep the physical meaning behind the coordinates.

BLEND LIGHT IN THE RIGHT SPACE

An sRGB value of 0.5 is not half the linear light of 1. Decode the transfer function, do light arithmetic in linear space, then encode for display. For an equal linear-light mix of black and white, the encoded sRGB result is about 0.735, not 0.5.

USE ORIENTATION WITH CARE

In two dimensions, the sign of $(b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$ identifies the side of a directed line. Near collinearity, floating rounding can change that sign. Robust geometric predicates need error analysis or adaptive precision, not an arbitrary epsilon pasted into every comparison.

ROTATIONS ARE NOT ORDINARY AVERAGES

Unit quaternions compactly represent 3D rotations. q and $-q$ represent the same rotation. Interpolation must account for that equivalence and maintain normalization. Simply averaging four components can choose an unintended path or produce a zero vector.

A USEFUL TINY CALCULATION

For a unit surface normal $n=(0,1,0)$ and a unit light direction $l=(0,0.6,0.8)$, $\max(0, n \cdot l) = 0.6$. If you omit normalization, the value also measures vector length, changing the intended lighting model.

USE THE MODEL DELIBERATELY

Checkpoint: identify units, frame, normalization and color space before asking which formula is fastest.

KEEP GOING / SPECIALTY GUIDE

Control, sensors and robotics

Uncertainty and timing are part of the state.

FUSE MEASUREMENTS WITH A MODEL

Two independent unbiased measurements of the same scalar quantity, with known positive variances v_1 and v_2 , can be combined using inverse-variance weights. For values 10 and 14 with variances 1 and 4, the fused value is $(10/1+14/4)/(1+1/4)=10.8$, and the fused variance is $1/(1+1/4)=0.8$.

CORRELATION CHANGES THE ANSWER

Sensors sharing a calibration bias are not independent. Treating them as independent can make the estimated uncertainty unrealistically small. Covariance matrices describe how errors move together; off-diagonal entries matter.

MAKE THE SAMPLE INTERVAL VISIBLE

A discrete PID uses $\text{integral} += \text{error} * dt$ and $\text{derivative} = (\text{error} - \text{previous}) / dt$. A zero or irregular dt changes the computation. Actuator saturation calls for an anti-windup strategy, and a derivative term often needs noise filtering.

USE A STABILITY QUESTION

A discrete linear update $x_{\text{next}} = a * x$ decays when $|a| < 1$. In several dimensions, the analogous linear-system condition concerns the eigenvalues of the update matrix. A controller that works in a slow simulation is not automatically safe on hardware.

USE THE MODEL DELIBERATELY

Checkpoint: transform every measurement to a common frame, record its uncertainty, and test the actual sampling and actuator limits.

KEEP GOING / SPECIALTY GUIDE

Compilers, protocols and information

Discrete structure turns large systems into explicit rules.

FIND A FIXED POINT

A live-variable analysis uses $\text{live_in} = \text{use} \cup (\text{live_out} - \text{def})$. Successor live_in sets form live_out . With monotone transfer on a finite set of possible facts, a correctly scheduled worklist converges. The result must respect the control-flow graph and boundary conditions.

DISTINGUISH ERROR DETECTION FROM IDENTITY

A CRC uses polynomial arithmetic over bits, where addition is XOR. It detects classes of accidental transmission errors under its specified polynomial and protocol. It does not authenticate the sender or resist deliberate modification; authentication needs an appropriate cryptographic construction and reviewed implementation.

READ ENTROPY AS AVERAGE INFORMATION

For possible outcomes with probabilities p , Shannon entropy is $-\sum(p \cdot \log_2(p))$, treating a zero-probability term as zero. A fair bit has one bit of entropy; a certain outcome has zero. This is not automatically a password-strength estimate or a guarantee of unpredictability.

SIZE A NETWORK PIPELINE

A 10 Gbit/s path with 40 ms round-trip time has bandwidth-delay product $10^9 \cdot 0.040/8 = 50,000,000$ bytes. That is a useful in-flight payload scale. Congestion control, loss, protocol overhead and receiver capacity can still limit throughput.

USE THE MODEL DELIBERATELY

Checkpoint: state the graph, arithmetic field, probability model or network boundary. The same symbol can represent very different systems.

KEEP GOING / SPECIALTY GUIDE

Money, time and deterministic arithmetic

Exactness begins with a domain rule.

STORE AN EXPLICIT UNIT

Integer cents represent multiples of one hundredth exactly while the integer stays in range. Not every currency or quantity uses two decimal places. Addition must still be checked, and tax or interest calculations need a rounding point and policy.

CHOOSE WHEN TO ROUND

Two line items of 5 cents taxed at 10% produce 0.5 cent tax each. With positive halfway-up rounding, rounding each line gives 2 cents total tax; summing first and rounding gives 1 cent. Neither expression defines the business policy by itself.

SEPARATE TIME CONCEPTS

A duration is an interval; a civil timestamp is a calendar label with a time-zone interpretation. Use a monotonic clock for elapsed-time measurements. Wall clocks can move after synchronization, and local calendar times can be ambiguous or nonexistent.

TREAT FIXED POINT AS A FORMAT

For raw integers a and b at scale 2^F , their product represents $a*b$ at scale $2^{(2F)}$. Multiply in a sufficiently wide type, rescale with a documented rounding rule, and check the destination range. Saturating arithmetic and wrapping arithmetic have different meanings.

USE THE MODEL DELIBERATELY

Checkpoint: write the rounding, overflow and time-zone policies beside the type definition so a future reader cannot guess them incorrectly.

ANSWERS / 01-06

Check your reasoning.

Compare the assumptions as well as the final number. A different boundary policy may produce a different valid answer.

01 / GIVE EVERY NUMBER A MEANING

$750 \times 1,048,576 \times 8 / 2,500,000,000 = 2.5165824$ seconds. Bits and bytes must be converted before dividing.

02 / TURN A FORMULA INTO A FUNCTION

$t = 15/60 = 0.25$, so $y = 1750$. For 80, extrapolation gives 4500; a clamping contract gives 4000; a validating contract rejects it.

03 / READ GROWTH WITH POWERS AND LOGS

20 bits distinguish up to 1,048,576 states. The next capacity is $8,192 = 2^{13}$.

04 / SEE BITS, THEN WRITE THE CONDITION

$0xA7 = 1010\ 0111$. The condition is `!ready || (!cached && !local)`.

05 / CHECK THE RANGE BEFORE MULTIPLYING

Addition requires $b \leq \text{SIZE_MAX} - a$. For power-of-two $A > 0$, first require $x \leq \text{SIZE_MAX} - (A - 1)$, then use $(x + A - 1) \& \sim(A - 1)$.

06 / REASON ABOUT CYCLES AND SCALES

The gcd is 21. Modular subtraction gives 10 ticks. Without the elapsed-time bound, 10, 266 and further values are indistinguishable.

ANSWERS / 07-12

Check your reasoning.

Compare the assumptions as well as the final number. A different boundary policy may produce a different valid answer.

07 / FLOATING POINT IS A MOVING RULER

No. These are 2^{53} and $2^{53}+1$. The next representable larger integer after 2^{53} is $2^{53}+2$.

08 / SPEND AN EXPLICIT ERROR BUDGET

$\log_{1p}$ avoids losing x when forming $1+x$. Extra arithmetic precision cannot restore information that the measurement never contained.

09 / A SUM IS A LOOP WITH A NAME

The sums are 10, 0 and 6. $[2,2)$ contains no elements, so its sum is zero.

10 / PROVE A LOOP IN ONE SENTENCE

It returns 0, 3 and 4. A maximum-scan invariant is: best equals the maximum of the nonempty prefix already processed.

11 / PREDICT WORK BEFORE MEASURING TIME

Unordered: $10 \times 9 \times 8 / (3 \times 2 \times 1) = 120$. Ordered: $10 \times 9 \times 8 = 720$. Constraints may eliminate some combinations.

12 / USE SETS AND GRAPHS TO EXPOSE STRUCTURE

10^{12} bits / 8 = 125,000,000,000 bytes, about 116.4 GiB before overhead. Adjacency lists are usually more appropriate for this sparse graph.

ANSWERS / 13-18

Check your reasoning.

Compare the assumptions as well as the final number. A different boundary policy may produce a different valid answer.

13 / COUNT OUTCOMES BEFORE USING BAYES

Both: $0.1 \times 0.1 = 0.01$. At least one: $1 - 0.9^2 = 0.19$. Shared causes can invalidate both independent-trial calculations.

14 / KEEP THE SHAPE OF THE DATA

The expectation is $.95 \times 4 + .05 \times 80 = 7.8$ ns. It says nothing by itself about long dependent sequences or a particular tail percentile.

15 / MEASURE UNCERTAINTY, NOT JUST SPEED

Four times as many, because standard error scales with $1/\sqrt{n}$. Repeating correlated observations does not automatically achieve this gain.

16 / TRADE MEMORY FOR A KNOWN ERROR RATE

About 819 under this model. Query distribution, hash quality and saturation can change the observed count.

17 / A VECTOR IS A LIST WITH GEOMETRY

The dot product is $4 + 0 - 3 = 1$. The perpendicular unit vectors have cosine zero.

18 / CHECK MATRIX SHAPES BEFORE VALUES

XW has shape 32 by 3. Each row produces three outputs, and every output uses an eight-term dot product.

ANSWERS / 19-24

Check your reasoning.

Compare the assumptions as well as the final number. A different boundary policy may produce a different valid answer.

19 / KEEP COORDINATES IN THEIR FRAMES

Translation changes an origin, while velocity describes change of position. Angular velocity is measured in radians per unit time, commonly rad/s.

20 / A DERIVATIVE IS LOCAL SENSITIVITY

$dy/dx = 2 \cos(2x)$. The approximate area error is $2\pi r dr$; relative area error is about twice relative radius error for nonzero r .

21 / TRAIN A MODEL BY REDUCING A LOSS

At 0.5 it reaches $w=3$ in one exact-arithmetic step. At 1 the error flips sign without shrinking. Above 1 its magnitude grows unless already at the optimum.

22 / ACCUMULATION TURNS RATES INTO TOTALS

$0 < dt < 0.2$. At $dt=0.2$ the multiplier is -1, so the sign alternates without decay in exact arithmetic.

23 / SIGNALS ARE WEIGHTED NEIGHBORHOODS

The output length is $4+3-1 = 6$. Use an FFT length of at least 6, often a convenient larger supported length.

24 / COUNT BYTES AS CAREFULLY AS OPERATIONS

The bandwidth-only minimum is 0.1 s. Cache-line size is a hardware property, not a universal C++ guarantee.

ANSWERS / 25-30

Check your reasoning.

Compare the assumptions as well as the final number. A different boundary policy may produce a different valid answer.

25 / PARALLEL WORK STILL HAS A SERIAL PART

Speedup is $1/((.1+.9)/8) = 4.706$. Deadlines also depend on release patterns, priorities, blocking, jitter and worst-case execution times.

26 / PREPARE DATA WITHOUT LEAKING THE ANSWER

The model sees different transformations, and validation has used information unavailable for an individual future input. Store and reuse the training parameters.

27 / TURN SCORES INTO A STABLE LOSS

Adding a common constant leaves the exact softmax unchanged. Multiplying changes concentration; with a unique maximum, a large positive multiplier puts more weight on it.

28 / BACKPROPAGATION IS THE CHAIN RULE REUSED

The gradients through the inactive ReLU are zero. At zero there is no unique derivative, so a finite-difference check may disagree with a chosen subgradient.

29 / SIMILARITY AND ATTENTION SHARE A DOT PRODUCT

QK^T is 5×7 . After row-wise normalization, multiplying by V gives a 5×3 output.

30 / MAKE QUANTIZATION AN ERROR-BUDGET DECISION

The bound is 0.01 in exact arithmetic. No: even a single product can exceed int8, and a sum requires a bound based on length and input ranges.

FORMULA ATLAS

Numbers and algorithms

UNITS

amount = rate x time

Use compatible units and distinguish bits from bytes.

AFFINE MAP

$t = (x-a)/(b-a); y = c + t(d-c)$

Require $a \neq b$; decide validation or clamping.

STATE COUNT

N Boolean flags $\rightarrow 2^N$ states

Constraints may reduce the reachable set.

CHECKED PRODUCT

$a \neq 0$ or $b \leq \text{MAX}/a$

For nonnegative integers, before multiplication.

CHECKED ADDITION

$b \leq \text{MAX} - a$

For unsigned a and b , before addition.

PREFIX QUERY

$\text{sum}[l, r] = P[r] - P[l]$

Require $0 \leq l \leq r \leq n$; totals must fit.

UNORDERED PAIRS

$n(n-1)/2$

Reduce before multiplying and check range.

GCD / LCM

$\text{lcm}(a, b) = (a/\text{gcd}(a, b)) * b$

For positive integers; the product must fit.

POWER-OF-TWO ALIGNMENT

$(x+A-1) \& \sim(A-1)$

$A > 0$ is a power of two; addition must fit.

ROW-MAJOR ELEMENT

$\text{offset} = \text{row} * \text{columns} + \text{column}$

Validate indices and checked total size.

FORMULA ATLAS

Uncertainty and performance

INDEPENDENT RETRIES

$$P(\text{any failure}) = 1 - (1-p)^n$$

p in $[0,1]$, n nonnegative; independence matters.

CONDITIONAL PROBABILITY

$$P(A|B) = P(A \text{ and } B)/P(B)$$

Require $P(B) > 0$; denominator is the known group.

EXPECTATION

$$E[X] = \text{sum}(\text{value} \times \text{probability})$$

Linearity does not require independence.

SAMPLE SPREAD

$$s^2 = \text{sum}((x - \text{mean})^2)/(n-1)$$

$n \geq 2$; stable evaluation matters.

MEAN UNCERTAINTY

$$SE \sim s/\sqrt{n}$$

Independent observations; finite variance.

BLOOM MODEL

$$p \sim (1 - \exp(-kn/m))^k$$

m bits, n keys, k probes; a hashing model.

FIXED-WORKLOAD SPEEDUP

$$1/(s + (1-s)/N)$$

s serial fraction; idealized parallel work.

LITTLE'S LAW

$$L = \text{arrival_rate} \times \text{mean_time}$$

Long-run stable averages, same boundary.

MEMORY CEILING

$$\text{operations/s} \leq \text{bytes/s} \times \text{operations/byte}$$

Traffic model and memory level must be stated.

BANDWIDTH-DELAY PRODUCT

$$\text{payload bytes} \sim \text{bit_rate} \times \text{RTT} / 8$$

Use seconds and bits/s; excludes extra effects.

FORMULA ATLAS

Geometry and learning

DOT PRODUCT

$$\mathbf{a} \cdot \mathbf{b} = \sum(\mathbf{a}[i] * \mathbf{b}[i])$$

Matching dimensions, frames and feature meaning.

COSINE SIMILARITY

$$(\mathbf{a} \cdot \mathbf{b}) / (|\mathbf{a}| |\mathbf{b}|)$$

Nonzero vectors; normalize robustly.

MATRIX SHAPE

$$(\mathbf{m} \times \mathbf{k})(\mathbf{k} \times \mathbf{n}) \rightarrow (\mathbf{m} \times \mathbf{n})$$

Each output is a k-term dot product.

GRADIENT STEP

$$\theta \leftarrow \theta - \text{rate} * \text{gradient}$$

Gradients from one consistent parameter state.

SQUARED LOSS

$$\mathbf{L} = \text{mean}((\text{prediction} - \text{target})^2)$$

Sensitive to large residuals; define units.

STABLE SOFTMAX

$$\exp(\mathbf{z}[i] - \max(\mathbf{z})) / \sum(\exp(\mathbf{z} - \max(\mathbf{z})))$$

Nonempty finite inputs; define extreme cases.

CLASS LOSS

$$\log(\sum(\exp(\mathbf{z} - \mathbf{m}))) - (\mathbf{z}[\text{target}] - \mathbf{m})$$

Avoid taking log of an underflowed probability.

CHAIN RULE

$$dy/dx = (dy/du)(du/dx)$$

For differentiable compositions.

EXPLICIT EULER

$$\mathbf{x}_{\text{next}} = \mathbf{x} + dt * \mathbf{f}(\mathbf{t}, \mathbf{x})$$

Old state in all derivatives; stability depends on dt.

AFFINE QUANTIZATION

$$\mathbf{q} = \text{clip}(\text{round}(\mathbf{x}/\mathbf{s}) + \mathbf{z}); \mathbf{x}_{\text{hat}} = \mathbf{s}(\mathbf{q} - \mathbf{z})$$

$s > 0$; specify range, rounding and accumulator.

GLOSSARY / A-O

Keep the meaning close.

Absolute error. Distance from a reference, measured in the same unit.

Amortized cost. Cost per operation averaged over an operation sequence with a deterministic total bound.

Bias. A systematic offset; in a model, also an additive parameter.

Conditioning. Sensitivity of the mathematical answer to changes in input.

Confidence interval. An interval-producing procedure with a stated repeated-sampling coverage under its assumptions.

Covariance. A measure of joint variation between two quantities.

Domain. The allowed inputs of a function.

Gradient. The vector of partial derivatives of a scalar function.

Invariant. A statement preserved at a specified point in a computation.

Logit. An unnormalized classification score before sigmoid or softmax.

Norm. A measure of vector magnitude satisfying the norm rules.

Overflow. A result outside the representable range of its arithmetic format.

GLOSSARY / P-Z

Keep the meaning close.

Percentile. A distribution position whose sample definition needs a convention.

Quantization. Mapping values to a discrete representable set.

Rank. The number of independent rows or columns of a matrix.

Residual. A difference such as prediction minus observation.

Sample. Observed values drawn or collected from a process or population.

Stability. How an algorithm controls numerical error; in dynamics, how perturbations evolve.

Tensor. In this book, an indexed multidimensional array used by numerical and learning software.

Underflow. A tiny floating result entering the subnormal range or rounding to zero, depending on the format and operation.

Unit. The scale and kind of a measured quantity.

Variance. Expected squared deviation from a mean, or a defined sample estimate.

Vectorization. Processing multiple elements with data-parallel operations.

Zero point. The integer code representing real zero in affine quantization.

Read deeper with a purpose.

These primary references support the language rules, statistical assumptions and AI formulas used in the relevant lessons. They are routes into deeper study, not prerequisites for this book.

R1 / C11 committee draft N1570

WG14, 2011. Integer promotions, expressions, remainder, shifts and the math library. This edition uses a C11 baseline, not a claim that C11 is the newest C standard.

<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

R2 / C++ working draft N4861

WG21, 2020. A public C++20-era draft: basic.fundamental, expr, numeric, cmath and library contracts. The labs deliberately use a C++20 baseline.

<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4861.pdf>

R3 / NIST/SEMATECH e-Handbook: confidence limits for the mean

Section 1.3.5.2. Mean intervals, sampling assumptions and the Student t multiplier.

<https://www.itl.nist.gov/div898/handbook/eda/section3/eda352.htm>

R4 / Accurately Computing the Log-Sum-Exp and Softmax Functions

P. Blanchard, D. J. Higham and N. J. Higham. IMA Journal of Numerical Analysis, 2021; preprint arXiv:1909.03469. Numerical behavior of shifted formulas.

<https://arxiv.org/abs/1909.03469>

R5 / Deep Learning

I. Goodfellow, Y. Bengio and A. Courville, MIT Press, 2016. Chapters 2-6 connect linear algebra, probability, numerical computation and learning.

<https://www.deeplearningbook.org/>

R6 / Attention Is All You Need

A. Vaswani and coauthors, 2017. Scaled dot-product attention and the Transformer architecture; arXiv:1706.03762.

<https://arxiv.org/abs/1706.03762>

R7 / Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

B. Jacob and coauthors, CVPR 2018; arXiv:1712.05877. Affine quantization and integer inference.

<https://arxiv.org/abs/1712.05877>

Source-to-topic map: R1-R2 -> Lessons 4-10 and library use; R3 -> Lesson 15; R4 -> Lesson 27; R5 -> Lessons 17-22 and 26-28; R6 -> Lesson 29; R7 -> Lesson 30.

What was simplified. What was checked.

A SHORTER ROUTE THROUGH THE FIRST EDITION

Original foundations and discrete topics. Lessons 1-6 and 9-12 combine overlapping ideas around concrete calculations.

Original machine arithmetic and numerical analysis. Lessons 5-8 and 20-24 keep representation, range and error beside the calculation.

Original probability and performance chapters. Lessons 13-16 and 24-25 link assumptions to measurement and capacity decisions.

Original geometry, calculus and specialty survey. Lessons 17-23 teach the common core; pages 80-83 retain focused bridges to specialized fields.

Expanded data and AI material. Lessons 26-30 and Projects C-D add preprocessing, losses, gradients, retrieval, attention and quantization.

CORRECTIONS AND CLEARER BOUNDARIES

The revision makes overflow checks explicit before multiplication and power-of-two growth; distinguishes exact endpoint equality from conditioning; states percentile conventions; distinguishes explicit from semi-implicit Euler; and avoids treating a fixed-count Newton loop as a general square-root routine.

VERIFICATION RECORD

All 30 lesson programs and four project programs were built and run with GCC/G++ 13.3.0 on Linux x86-64, using C11 or C++20, optimization and warning flags. Output checks and dedicated boundary/invariant checks are included with the source. Additional undefined-behavior checks were run on the numerical boundary tests.

Windows, macOS, MSVC and other processors were not executed in this environment. The examples avoid platform APIs, but compiler support and final floating-point digits can differ. Tests establish the checked cases; they do not prove unrestricted numerical correctness.

A TRANSPARENT SCOPE

This is a practical learning edition. It introduces the mathematics needed to reason about larger libraries and systems; it does not implement an industrial BLAS, FFT, geometry kernel, cryptographic library or complete learning framework.

Build confidence by changing one thing.

You understand a formula when you can predict what happens after its assumptions or inputs change.

RELIABLE C/C++ ARITHMETIC

Read 1-10. Explain every type, unit, input range and failure path. Modify a buffer-size calculation so it rejects an overflowing product.

DATA ANALYSIS AND MEASUREMENT

Read 13-16 and Project A. Collect a small dataset, choose a percentile convention and report both a typical value and the tail.

GEOMETRY, GRAPHICS AND SIMULATION

Read 17-23 and Project B. Draw the coordinate frame, choose a boundary rule and compare a numerical result against a tiny exact case.

PRACTICAL AI MATHEMATICS

Read 17-21 and 26-30, then Projects C-D. Verify one gradient numerically, keep preprocessing tied to training data and explain the meaning of every score.

PERFORMANCE ENGINEERING

Read 11-12 and 24-25. Predict growth, count bytes, state the parallel fraction and measure uncertainty before announcing a speedup.

THE HABIT THAT LASTS

Before trusting a result, ask: What does this number mean? Which assumptions make it valid? What tiny case can I check independently? What changes at the boundary?

Ayman Alheraki

NO-FEAR MATH

**An idea you can picture.
A result you can explain.
Code you can inspect.**

Make mathematics a practical part of programming: from a safe byte count and a trustworthy average to vectors, learning, attention and quantization.

30 visual lessons

34 complete C and C++ programs

4 worked projects

Exercises, explanations and a compact formula atlas

Prepared by Ayman Alheraki

REVISED SECOND EDITION / OCTOBER 2026