

ROBOTICS COURSE

Programming for Robotics with Arduino C++

12 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	Robot Software Architecture	2
Chapter 2	Talking to Hardware	25
Chapter 3	A Simulator to Test In	54
Chapter 4	Sensor Code and Filtering	76
Chapter 5	Driving	110
Chapter 6	Control Loops	133
Chapter 7	State Machines	155
Chapter 8	Interrupts and Timing	182
Chapter 9	Planning in Kilobytes	202
Chapter 10	State Estimation	228
Chapter 11	Robot Arms	249
Chapter 12	Testing and Safety	274

Programming for Robotics with Arduino C++

Course 2 · Chapter 1 · Robot Software Architecture

Programming for Robotics 1 (the Python course) built every robot idea on a computer with a gigabyte of memory and a clock nobody thought about. This course builds the same twelve chapters again for the kind of computer that sits *inside* a robot: an **Arduino Uno R3**, an 8-bit AVR chip with **2 KB of RAM**, **32 KB of flash** for the program, 16 MHz, and **no floating-point hardware**. Nothing here is slower than it needs to be by accident, and nothing is hidden: you will see what a log costs in bytes, what a mistake in a timer does to a robot, and why a habit that is harmless on a laptop (`delay()`, a `String`, a virtual function) is not harmless on a board.

The chapter assumes **no C++**. The language is taught in boxes as it is needed, and section 2 is a short tour for someone who knows Python. The robot is a very small one on purpose: it drives toward a wall and stops 0.25 m short of it. The interesting part is the *architecture* around it: how time is kept, how tasks are scheduled, how code is split into files and how the board's memory is budgeted.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`, no warnings): every block below that has an output under it. These programs use a *fake clock* (an integer that the program itself advances), so their millisecond figures are exact and repeatable, but they are **simulations of timing, not measurements of a board**. The headers (`timing.h`, `scheduler.h`, `log.h`, `robot.h`) contain no Arduino calls, so the very same text is compiled both for the PC and for the board.
> **Compiled for the Uno, not run** (arduino-cli 1.6.0, core `arduino:avr` 1.8.8): the sketch `robot_ch1.ino`. The “Sketch uses” and “Global variables use” figures in section 11 are the real compiler's reports; they are *static* sizes. **No board was attached**, so no timing, no serial output and no stack depth on a real Uno is claimed anywhere in this chapter.

1. What changes when the computer is a chip

The robot code of the Python course was the same in spirit: read a sensor, decide, command a motor, repeat at a fixed rate. What changes is the setting.

	PYTHON ON A PC (PROGRAMMING FOR ROBOTICS 1)	C++ ON AN UNO

Memory	gigabytes; a list grows as needed	2048 bytes of RAM; every array has a size fixed when the program is built
Program space	effectively unlimited	32,256 bytes of flash after the bootloader
Numbers	`int` never overflows; `float` is a 64-bit double	`int` is 16 bits and *wraps*; `double` is only 32 bits (section 2)
Errors	an exception, with a traceback	usually nothing: a wrong index or a stack that runs out silently corrupts something else
Time	an operating system schedules you; `time.sleep()` is polite	nobody schedules you; your `loop()` *is* the operating system
Checking	run it and look	the compiler reports memory use before it ever runs: use it (section 11)

2. C++ for Python programmers

Eight ideas carry most of what this chapter needs. Each is shown by running something.

1 · TYPES ARE DECLARED AND HAVE A SIZE

Python's `x = 7` works out the type at run time; C++ fixes it at compile time (`int x = 7;`, `float y = 0.5f;`, `uint8_t n = 200;`). The size depends on the machine, and that is the first trap: the program below, run on the PC, prints the PC's sizes, and the sketch of section 11 *proves* the board's with `static_assert` (a check the compiler makes while building, with no cost at run time).

```
// e_types.cpp -- what C++ does that Python does not (host build; the board's sizes are
#include <cstdio>
#include <cstdint>

int main() {
  std::printf("host sizes in bytes: int %zu, long %zu, long long %zu, float %zu, double %
              sizeof(int), sizeof(long), sizeof(long long), sizeof(float), sizeof(double
  uint8_t a = 250;
  a = (uint8_t)(a + 10);                                  // a uint8_t holds 0..255 and
  std::printf("uint8_t 250 + 10 = %d\n", a);
  unsigned int u = 0;
  u = u - 1;                                              // unsigned arithmetic is modu
  std::printf("unsigned 0 - 1 = %u\n", u);
```

```
  std::printf("7 / 2 = %d    7 / 2.0f = %.2f    -7 / 2 = %d    -7 %% 3 = %d\n", 7 / 2, 7
  float big = 16777216.0f;                                  // 2^24: the end of the float
  std::printf("float 16777216 + 1 = %.1f    (float has a 24-bit mantissa)\n", big + 1.0f
  float tenth = 0.1f, sum = 0.0f;
  for (int i = 0; i < 10; ++i) sum += tenth;
  std::printf("0.1f is stored as %.10f;  ten of them sum to %.10f\n", tenth, sum);
  int truncated = (int)-3.9f;
  std::printf("(int)-3.9 = %d  (it truncates toward zero)\n", truncated);
  return 0;
}
```

```
host sizes in bytes: int 4, long 4, long long 8, float 4, double 8, pointer 8
uint8_t 250 + 10 = 4
unsigned 0 - 1 = 4294967295
7 / 2 = 3    7 / 2.0f = 3.50    -7 / 2 = -3    -7 % 3 = -1
float 16777216 + 1 = 16777216.0    (float has a 24-bit mantissa)
0.1f is stored as 0.1000000015;  ten of them sum to 1.0000001192
(int)-3.9 = -3  (it truncates toward zero)
```

The same questions in Python, for comparison:

```
# the same lines in Python
print("7 // 2 =", 7 // 2, "   7 / 2 =", 7 / 2, "   -7 // 2 =", -7 // 2, "   -7 % 3 =", -
print("250 + 10 =", 250 + 10, "  (an int never wraps)    2**24 + 1.0 =", 2**24 + 1.0, "
print("int(-3.9) =", int(-3.9))
```

```
7 // 2 = 3    7 / 2 = 3.5    -7 // 2 = -4    -7 % 3 = 2
250 + 10 = 260   (an int never wraps)    2**24 + 1.0 = 16777217.0   (a Python float is a
int(-3.9) = -3
```

TYPE	THIS PC (RUN)	UNO R3 (COMPILE-TIME CHECKED)
`int`	4 bytes	**2 bytes** (so 32767 + 1 wraps)
`long`	4 bytes	4 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes

`double`	8 bytes	**4 bytes: the same as float**
pointer	8 bytes	2 bytes

Three results in the output matter for the rest of the chapter: `uint8_t` and `unsigned` arithmetic **wrap around** (`250 + 10` is 4; `0 - 1` is 4294967295), integer division **truncates toward zero** (`-7 / 2` is -3, where Python's `//` floors to -4), and a 32-bit `float` cannot count past 16,777,216 (`+ 1` changes nothing). Python never does any of these to an `int`.

2 · CLASSES, AND A HEADER FILE INSTEAD OF AN IMPORT

A `class` bundles data with the functions that use it, as in Python, but its members are declared with a type and an access level (`public:` / `private:`). C++ has no `import`; a `.h` *header* is pasted into the file by `#include`, and `#pragma once` at the top stops it being pasted twice. The headers of this chapter hold their code in full, which is the simple way to share code in a small Arduino project.

3 · REFERENCES, POINTERS AND `const`

`HW& hw` is a *reference*: the object itself, not a copy (what every Python variable already is). `void* ctx` is a *pointer*: the address of something, used where a plain function must be handed an object. `const` on a function (`int size() const`) promises that it changes nothing.

4 · TEMPLATES: CODE GENERATED FOR A TYPE

`template <class HW> class WallRobot` is a recipe; `WallRobot<SimHW>` makes the compiler write a copy of the class for that one hardware type, calls and all. A Python class works with any object that has the right methods, decided while the program runs; a template gets the same freedom decided *while compiling*, so it costs nothing at run time (section 8 measures this). `template <uint8_t N>` takes a *number*, which is how a buffer gets its size.

The rest (the `setup()` / `loop()` pair, `millis()`, `Serial`, `F("...")`) are Arduino's, and appear where they are first used.

3. The first loop: sense, think, act

An Arduino program is two functions. `setup()` runs once. `loop()` is then called again and again, forever, as fast as it can run. There is no `main()` to write and no `while True:`: the loop is the board's own.

The robot is three small methods on a class, the same sense-think-act shape as the robots of Programming for Robotics 1: **sense** reads the world (a range), **think** turns the range into a speed with a proportional controller limited to a top speed, **act** writes the speed. The hardware is a template parameter, so the same robot runs against a simulated world on the PC and the board:

```
// robot.h -- the sense-think-act pieces of a robot that drives toward a wall and stops (
// The hardware is a TEMPLATE parameter HW with two functions, range_m() and drive(speed
// abstraction costs no RAM and no time (section 8 measures a virtual-function version fo
#pragma once
#include <stdint.h>

inline float clampf(float v, float lo, float hi) { return v < lo ? lo : (v > hi ? hi : v

template <class HW>
class WallRobot {
 public:
  WallRobot(HW& hw, float stop_m, float gain, float vmax_mps)
      : hw_(hw), stop_m_(stop_m), gain_(gain), vmax_(vmax_mps), range_(0.0f), cmd_(0.0f)

  void sense(uint32_t) { range_ = hw_.range_m(); }                                  // S
  void think(uint32_t) { cmd_ = clampf(gain_ * (range_ - stop_m_), 0.0f, vmax_); }   // T
  void act(uint32_t) { hw_.drive(cmd_); }                                           // A

  float range() const { return range_; }
  float command() const { return cmd_; }

  // Scheduler trampolines: a TaskFn is a plain function pointer, so the object travels
  static void sense_task(void* c, uint32_t now) { static_cast<WallRobot*>(c)->sense(now)
  static void think_task(void* c, uint32_t now) { static_cast<WallRobot*>(c)->think(now)
  static void act_task(void* c, uint32_t now) { static_cast<WallRobot*>(c)->act(now); }

 private:
  HW& hw_;
  float stop_m_, gain_, vmax_, range_, cmd_;
};

// A simulated world for the PC and for the board: a wall 1.00 m ahead, a robot that move
struct SimHW {
  float x_m, speed_mps, wall_m;
  explicit SimHW(float wall = 1.0f) : x_m(0.0f), speed_mps(0.0f), wall_m(wall) {}
  float range_m() { return wall_m - x_m; }
  void drive(float v) { speed_mps = v; }
```

```
  void advance(float dt_s) { x_m += speed_mps * dt_s; }
};
```

Run for six simulated seconds with all three at 20 ms (50 Hz), against a wall 1.00 m away:

```
// e_robot.cpp -- the wall robot in closed loop: sense, think and act every 20 ms, a wor
#include <cstdio>
#include "scheduler.h"
#include "robot.h"

int main() {
  SimHW world(1.0f);                                        // a wall 1.00 m ahead, the r
  WallRobot<SimHW> robot(world, 0.25f, 1.5f, 0.5f);         // stop 0.25 m short; gain 1.
  Scheduler<3> sch;
  sch.add(WallRobot<SimHW>::sense_task, &robot, 20);
  sch.add(WallRobot<SimHW>::think_task, &robot, 20);
  sch.add(WallRobot<SimHW>::act_task, &robot, 20);
  std::printf("  t s   range m   speed m/s\n");
  for (uint32_t t = 0; t <= 6000; ++t) {
    sch.run(t);
    world.advance(0.001f);
    if (t % 500 == 0) std::printf("  %3.1f   %7.3f   %8.3f\n", t / 1000.0, world.range_m
  }
  std::printf("at 6 s the robot is %.3f m from the wall (the stop line is 0.250 m); sens
              world.range_m(), (unsigned)sch.task(0).timer.runs());
  return 0;
}
```

```
  t s   range m   speed m/s
  0.0     0.999      0.500
  0.5     0.750      0.500
  1.0     0.508      0.388
  1.5     0.371      0.181
  2.0     0.306      0.085
  2.5     0.276      0.040
  3.0     0.262      0.018
  3.5     0.256      0.009
  4.0     0.253      0.004
  4.5     0.251      0.002
  5.5     0.250      0.000
```

```
   6.0     0.250      0.000
at 6 s the robot is 0.250 m from the wall (the stop line is 0.250 m); sense, think and a
```

The robot drives at the top speed, slows as the error shrinks, and settles on the stop line at 0.250 m (0.253 at 4 s; 0.250 by 5.5 s). It is a first-order approach: each second removes about the same fraction of what is left. All three tasks ran 301 times: once per 20 ms plus the first call.

4. Time on a microcontroller, and the wrap-around

`millis()` returns the milliseconds since power-on as an `unsigned long`, 32 bits. It is a number that **wraps to 0** after 2^{32} ms, which is 49.71 days; `micros()` does the same after 2^{32} microseconds, which is only 71.58 minutes (both figures are the first line of the output below). The wrap is not an unlikely event: it is certain, and a robot that is left switched on will meet it. The standard code that *looks* right is `if (now >= next)`; the correct code subtracts: `now - then` is computed modulo 2^{32}, so it gives the right elapsed time even when `now` has wrapped past `then`. The header holds both, the wrong one only so it can be shown failing, and a drift-free timer built on the correct one:

```
// timing.h -- time on a microcontroller (Programming for Robotics with Arduino C++ 1). I
#pragma once
#include <stdint.h>

// Time is an unsigned 32-bit counter that WRAPS: millis() after 2^32 ms (49.7 days), mi
// In C++ unsigned arithmetic is defined modulo 2^32, so this subtraction is correct ever
inline uint32_t elapsed(uint32_t now, uint32_t then) { return now - then; }

// WRONG, and very common: it compares two times as if the counter never wrapped. It is I
inline bool naive_due(uint32_t now, uint32_t next) { return now >= next; }

// A fixed-rate timer that does not drift and does not burst. due(now) is true at most or
class Periodic {
 public:
  explicit Periodic(uint32_t period) : period_(period), last_(0), started_(false), runs_

  uint32_t lateness(uint32_t now) const {              // how long after its moment the t
    if (!started_) return 0;
    uint32_t dt = elapsed(now, last_);
    return dt >= period_ ? dt - period_ : 0;
  }
```

```
  bool due(uint32_t now) {
    if (!started_) { started_ = true; last_ = now; ++runs_; return true; }
    uint32_t dt = elapsed(now, last_);
    if (dt < period_) return false;
    if (dt < 2 * period_) {
      last_ += period_;                              // on time, or a little late: ste
    } else {
      uint32_t n = dt / period_;                     // late by whole periods: stay on
      last_ += n * period_;
      skipped_ += n - 1;
    }
    ++runs_;
    return true;
  }

  uint32_t period() const { return period_; }
  uint32_t runs() const { return runs_; }
  uint32_t skipped() const { return skipped_; }

 private:
  uint32_t period_, last_;
  bool started_;
  uint32_t runs_, skipped_;
};
```

A fake clock is started a few milliseconds before the wrap, and a 5 ms timer is run both ways. `start = 4294967290` is 6 ms before the wrap; `4294967293` is 3 ms before it:

```
// e_wrap.cpp -- a 5 ms timer across the wrap of the millisecond counter, done right and
#include <cstdio>
#include "timing.h"

static void run(uint32_t start, bool verbose) {
  Periodic good(5);
  uint32_t next = start;                                 // the naive timer: "run when
  int good_n = 0, naive_n = 0;
  uint32_t gl = 0, nl = 0, gmin = 999, gmax = 0, nmin = 999, nmax = 0;   // last run tim
  bool gseen = false, nseen = false;
  if (verbose) std::printf("  k   now          Periodic   naive\n");
  for (uint32_t k = 0; k <= 30; ++k) {
    uint32_t now = start + k;                            // wraps to 0, 1, 2 ... in un
    bool g = good.due(now);
```

```
    bool n = naive_due(now, next);
    if (n) next += 5;
    good_n += g;
    naive_n += n;
    if (g) { if (gseen) { uint32_t d = now - gl; if (d < gmin) gmin = d; if (d > gmax) g
    if (n) { if (nseen) { uint32_t d = now - nl; if (d < nmin) nmin = d; if (d > nmax) n
    if (verbose && (k <= 8 || g || n))
      std::printf("%3u   %-11u  %-8s   %s\n", (unsigned)k, (unsigned)now, g ? "RUN" : ""
  }
  std::printf("start = %u (%u ms before the wrap): Periodic ran %d times, gaps %u..%u ms
              (unsigned)start, (unsigned)(0u - start), good_n, (unsigned)gmin, (unsigned
}

int main() {
  std::printf("millis() wraps after %.2f days; micros() after %.2f minutes\n", 429496729
  run(4294967290u, false);                                  // 6 ms before the wrap: by l
  run(4294967293u, true);                                   // 3 ms before the wrap: next
  std::printf("elapsed(2, 4294967290) = %u  (the true gap is 8 ms)\n", (unsigned)elapsed
  return 0;
}
```

```
millis() wraps after 49.71 days; micros() after 71.58 minutes
start = 4294967290 (6 ms before the wrap): Periodic ran 7 times, gaps 5..5 ms; naive ran
  k   now          Periodic   naive
  0   4294967293   RUN        RUN
  1   4294967294              RUN
  2   4294967295              RUN
  3   0
  4   1
  5   2            RUN
  6   3
  7   4
  8   5
 10   7            RUN
 15   12           RUN        RUN
 20   17           RUN        RUN
 25   22           RUN        RUN
 30   27           RUN        RUN
start = 4294967293 (3 ms before the wrap): Periodic ran 7 times, gaps 5..5 ms; naive ran
elapsed(2, 4294967290) = 8  (the true gap is 8 ms)
```

Read the first line carefully, because it is the more dangerous one: **6 ms before the wrap the naive timer works perfectly** (gaps 5..5), because the timer's target and the clock wrap in step. A test that happened to start there would pass. **3 ms before the wrap** it fails: the target (4294967298) wrapped to 2 while the clock was still at 4294967293, so `now >= next` is true on three consecutive passes (k = 0, 1, 2 in the table), and then the timer stays silent for 12 ms. Its *count* of runs is still 7, the same as the correct timer's; only the gaps (1..13 ms, against 5..5) show the fault. Judge a timer by the gaps between its runs, not by how many times it ran.

Challenge 1 shows how bad this is: of 40 possible start offsets, the naive timer misbehaves for 37. `elapsed()`, the last line of the output, is the whole cure: `elapsed(2, 4294967290)` is 8, the true gap, with no special case for the wrap.

5. A fixed rate without drift

The loop most beginners write is `work(); delay(20);`. It *aims* at 50 Hz, but each pass takes the work plus 20 ms, so the rate is lower and, worse, wanders with the work. A `Periodic` timer instead keeps a schedule of moments (0, 20, 40 ms ...) and asks each pass whether the next moment has arrived; it advances by exactly one period, never to "now", which is what removes drift. If a pass is late by whole periods, it stays on the grid, counts the periods it missed (`skipped`) and runs *once*, not in a burst.

```
// e_drift.cpp -- delay(20) after the work, against a Periodic timer, with 1 to 6 ms of w
#include <cstdio>
#include <random>
#include <algorithm>
#include "timing.h"

struct Stats {
  uint32_t n = 0, last = 0, lo = 0xFFFFFFFFu, hi = 0;
  double sum = 0;
  void tick(uint32_t t) {
    if (n > 0) { uint32_t d = t - last; lo = std::min(lo, d); hi = std::max(hi, d); sum +
    ++n;
    last = t;
  }
};

int main() {
  const uint32_t seconds = 10, end_ms = seconds * 1000;
```

```
  std::mt19937 rng(1);
  std::uniform_int_distribution<int> work(1, 6);          // ms of work per pass, diffe

  Stats a;                                                // loop 1:  work(); delay(20)
  for (uint32_t t = 0; t < end_ms;) {
    a.tick(t);
    t += work(rng);                                       // the work
    t += 20;                                              // delay(20)
  }

  rng.seed(1);
  Stats b;                                                // loop 2:  if (timer.due(now
  Periodic timer(20);
  for (uint32_t t = 0; t < end_ms; ++t) {
    if (timer.due(t)) {
      b.tick(t);
      t += work(rng) - 1;                                 // the work uses the clock up
    }
  }
  std::printf("10 s of a nominal 50 Hz loop, work 1 to 6 ms per pass:\n");
  std::printf("  loop                passes   mean ms   shortest   longest\n");
  std::printf("  work(); delay(20)    %6u   %7.2f   %8u   %7u\n", (unsigned)a.n, a.sum /
  std::printf("  Periodic(20)         %6u   %7.2f   %8u   %7u   (skipped %u)\n", (unsign
  return 0;
}
```

```
10 s of a nominal 50 Hz loop, work 1 to 6 ms per pass:
  loop                passes   mean ms   shortest   longest
  work(); delay(20)       428     23.42         21        26
  Periodic(20)            500     20.00         20        20   (skipped 0)
```

With 1 to 6 ms of work per pass, the `delay(20)` loop managed 428 passes in 10 s instead of 500: a mean of 23.42 ms, intervals anywhere from 21 to 26 ms. A controller that assumes a 20 ms step is wrong by 17 % on average here and by a different amount every pass. The `Periodic` loop ran 500 passes at exactly 20.00 ms, none skipped. (The work is the same random sequence in both loops, and the “work” here is a fake clock jump, so these are exact for the model and say nothing about how long a real servo write takes.)

6. Never block

`delay(500)` is the most dangerous function in the Arduino toolbox for a robot, because it does *nothing* for 500 ms: no sensor is read, no button is seen, no emergency stop is noticed. A robot that blinks a light with `delay` cannot also watch for a bump. The cost, measured on a button pressed at 20 random times:

```
// e_block.cpp -- how long does a button press wait to be noticed? 20 presses at random
#include <cstdio>
#include <random>
#include <vector>
#include <algorithm>
#include "timing.h"

int main() {
  std::mt19937 rng(7);
  std::uniform_int_distribution<uint32_t> when(500, 19500);
  std::vector<uint32_t> presses;
  for (int i = 0; i < 20; ++i) presses.push_back(when(rng));
  std::sort(presses.begin(), presses.end());

  // Version 1: blocking.   loop() { led(HIGH); delay(500); led(LOW); delay(500); if (bu
  double sum1 = 0; uint32_t max1 = 0;
  for (uint32_t p : presses) {
    uint32_t next_look = ((p / 1000) + 1) * 1000;          // the end of the current 100
    uint32_t lat = next_look - p;
    sum1 += lat; max1 = std::max(max1, lat);
  }
  // Version 2: non-blocking; the loop spins and looks at the button every pass (about 1
  double sum2 = 0; uint32_t max2 = 0;
  for (size_t i = 0; i < presses.size(); ++i) { uint32_t lat = 1; sum2 += lat; max2 = st
  // Version 3: non-blocking, the button polled by a Periodic(10) timer.
  double sum3 = 0; uint32_t max3 = 0;
  for (uint32_t p : presses) {
    Periodic poll(10);
    uint32_t t = 0;
    while (true) { if (poll.due(t) && t >= p) break; ++t; }
    uint32_t lat = t - p;
    sum3 += lat; max3 = std::max(max3, lat);
  }
  std::printf("20 presses; time from the press to the program noticing it (ms):\n");
  std::printf("  version                                    mean    worst\n");
  std::printf("  blocking: delay(500) twice per pass      %5.1f   %6u\n", sum1 / 20, (un
  std::printf("  non-blocking, polled every pass (1 ms)   %5.1f   %6u\n", sum2 / 20, (un
  std::printf("  non-blocking, Periodic(10) poll          %5.1f   %6u\n", sum3 / 20, (un
```

```
  return 0;
}
```

```
20 presses; time from the press to the program noticing it (ms):
  version                                  mean    worst
  blocking: delay(500) twice per pass     496.1      979
  non-blocking, polled every pass (1 ms)    1.0        1
  non-blocking, Periodic(10) poll           3.6        9
```

With the blocking blink the press waited a mean of 496.1 ms and as much as 979 ms before the program looked. Polling every pass noticed in 1 ms (that row is *assumed* to be one pass of a 1 ms loop, not measured). Polling from a 10 ms timer averaged 3.6 ms and never took more than 9. The cure is not clever: replace every `delay` with a timer and let `loop()` spin, so that the time between looks at the sensors is short and bounded.

7. A static scheduler

Once there are several jobs at several rates, one `Periodic` each, checked in turn, is a *scheduler*. This one has a fixed table of tasks (a plain array, size known at compile time, no heap), each a function pointer, a context pointer and a timer. Tasks earlier in the table are looked at first, so they have priority when two are due at once. It records each task's **worst lateness**, which is the number a real-time programmer actually wants.

```
// scheduler.h -- a cooperative scheduler with a fixed task table: no heap, no threads, n
#pragma once
#include <stdint.h>
#include "timing.h"

typedef void (*TaskFn)(void* ctx, uint32_t now);

struct Task {
  TaskFn fn;
  void* ctx;
  Periodic timer;
  uint32_t worst_late;                              // the largest lateness at which t
  Task() : fn(0), ctx(0), timer(1), worst_late(0) {}
};

// N is the capacity, fixed at compile time: the table is a plain array, so its RAM cost
```

```
template <uint8_t N>
class Scheduler {
 public:
  Scheduler() : n_(0) {}

  bool add(TaskFn fn, void* ctx, uint32_t period) {    // earlier tasks have priority: t
    if (n_ >= N) return false;
    tasks_[n_].fn = fn;
    tasks_[n_].ctx = ctx;
    tasks_[n_].timer = Periodic(period);
    ++n_;
    return true;
  }

  void run(uint32_t now) {                             // call this from loop(); each ta
    for (uint8_t i = 0; i < n_; ++i) {
      Task& t = tasks_[i];
      uint32_t late = t.timer.lateness(now);
      if (t.timer.due(now)) {
        if (late > t.worst_late) t.worst_late = late;
        t.fn(t.ctx, now);
      }
    }
  }

  uint8_t size() const { return n_; }
  const Task& task(uint8_t i) const { return tasks_[i]; }

 private:
  Task tasks_[N];
  uint8_t n_;
};
```

Three tasks (sense every 10 ms costing 1 ms, control every 20 ms costing 3 ms, a log every 100 ms costing 2 ms) over one simulated second, then the same with a control task that takes 35 ms, longer than its own period:

```
// e_sched.cpp -- a Scheduler with three tasks over one simulated second, then the same w
#include <cstdio>
#include "scheduler.h"

static uint32_t g_now = 0;                                 // the fake clock; a task tha
```

```
struct Counter { const char* name; uint32_t cost_ms; uint32_t calls; };

static void work(void* ctx, uint32_t) { Counter* c = static_cast<Counter*>(ctx); ++c->ca

static void run(uint32_t control_cost) {
  Counter sense = {"sense   10 ms", 1, 0}, control = {"control 20 ms", control_cost, 0},
  Scheduler<4> sch;
  sch.add(work, &sense, 10);
  sch.add(work, &control, 20);
  sch.add(work, &log, 100);
  for (g_now = 0; g_now < 1000; ++g_now) sch.run(g_now);
  std::printf("  task            period   runs   skipped   worst lateness ms\n");
  Counter* all[3] = {&sense, &control, &log};
  for (uint8_t i = 0; i < sch.size(); ++i)
    std::printf("  %-14s  %5u   %4u   %7u   %8u\n", all[i]->name, (unsigned)sch.task(i).
                (unsigned)sch.task(i).timer.skipped(), (unsigned)sch.task(i).worst_late)
}

int main() {
  std::printf("tasks cost 1, 3 and 2 ms (control takes 3 ms):\n");
  run(3);
  std::printf("the same, but control now takes 35 ms, more than its own 20 ms period:\n"
  run(35);
  return 0;
}
```

```
tasks cost 1, 3 and 2 ms (control takes 3 ms):
  task            period   runs   skipped   worst lateness ms
  sense   10 ms      10    100         0          0
  control 20 ms      20     50         0          0
  log    100 ms     100     10         0          0
  task            period   runs   skipped   worst lateness ms
  sense   10 ms      10     27        72         37
  control 20 ms      20     27        23         36
  log    100 ms     100     10         0         30
```

With honest costs everything runs on time: 100, 50 and 10 runs, nothing skipped, lateness 0. With the 35 ms task, *every* task suffers: sense should have run 100 times and ran 27, with 72 periods skipped; control itself ran 27 times with 23 skipped; and even the 100 ms log, which skips none, started up to 30 ms late. A cooperative scheduler cannot interrupt a task that is

running, so one slow task spoils the timing of all the others. This is why the rule of the rest of the course is that **a task does a small amount of work and returns**.

8. Templates against virtual functions

The robot reaches the hardware through `HW`. There are two ways to do that. A *template* parameter (as in `robot.h`) lets the compiler build the call directly. A *virtual* interface (an abstract base class with `virtual` functions, as Python programmers expect) makes every call go through a table of function pointers, looked up while running. Both were compiled for the Uno, with everything else identical:

VERSION	FLASH (OF 32,256)	RAM (OF 2,048)
template `WallRobot<SimHW>`	4950 bytes	473 bytes
virtual `WallRobot<IHW>`	5002 bytes (+52)	483 bytes (+10)

The virtual version costs 52 more bytes of flash and 10 more bytes of RAM (compile-time figures; the compiler's report does not say which part of the 10 is the object's pointer and which is the table). On a board with 2 KB that is half a percent of the RAM, so it is not a reason to avoid virtual functions in a place where they make the code clearer. It also does not show how much *time* each call costs, because the program was not run on a board. The honest rule: **measure, do not guess**; and a template is the default because its cost is zero by construction.

9. Project layout

An Arduino sketch is a folder; the `.ino` file has the same name as the folder, and every other file in the folder is compiled with it. Keeping the pure logic in headers with no Arduino calls is what lets the PC test the code that runs on the robot:

```
robot_ch1/
├── robot_ch1.ino      setup() and loop(); the only file that calls Arduino functions
├── timing.h           elapsed(), Periodic   (plain C++, builds on a PC)
├── scheduler.h        Task, Scheduler<N>    (plain C++)
├── log.h              LogEntry, RingLog<N>  (plain C++)
└── robot.h            WallRobot<HW>, SimHW (plain C++)
```

The result is that every number in sections 3 to 7 and 10 came from compiling these same headers with g++ on a PC, which can be done in a second with strict warnings, and they were then compiled unchanged for the Uno.

10. Logging: a ring buffer of codes

A robot has no screen and, once it moves, no cable. Its record of what happened is a **black-box log**: a fixed-size buffer of small records where the newest entry overwrites the oldest when it is full, so the log always holds the *last* N events, which are the ones that matter after a crash. A message is stored as a one-byte **code**, not a string, because on this board each character of a string costs a byte of RAM. A record is 8 bytes: a 4-byte time, a level, a code and a 16-bit value.

```
// log.h -- a black-box log: a ring buffer of small fixed-size records. Messages are COD
#pragma once
#include <stdint.h>

enum LogLevel : uint8_t { LOG_DEBUG = 0, LOG_INFO = 1, LOG_WARN = 2, LOG_FAULT = 3 };

struct LogEntry {                                    // 8 bytes on the board and on th
  uint32_t t;
  uint8_t level;
  uint8_t code;
  int16_t value;
};

template <uint8_t N>
class RingLog {
  static_assert(N > 0, "RingLog capacity must be 1..255: a larger value was truncated to
 public:
  RingLog() : head_(0), count_(0), dropped_(0) {}

  void add(uint32_t t, LogLevel level, uint8_t code, int16_t value) {
    if (count_ == N) ++dropped_; else ++count_;        // when full, the oldest record i
    buf_[head_].t = t;
    buf_[head_].level = level;
    buf_[head_].code = code;
    buf_[head_].value = value;
    head_ = (uint8_t)((head_ + 1) % N);
  }

  uint8_t size() const { return count_; }
```

```
  uint32_t dropped() const { return dropped_; }
  const LogEntry& at(uint8_t i) const {               // i = 0 is the oldest record sti
    uint8_t first = (uint8_t)((head_ + N - count_) % N);
    return buf_[(first + i) % N];
  }

 private:
  LogEntry buf_[N];
  uint8_t head_, count_;
  uint32_t dropped_;
};
```

Eight records, twelve events:

```
// e_log.cpp -- a RingLog of 8 records, 12 events added: the oldest 4 are overwritten an
#include <cstdio>
#include "log.h"

static const char* const LEVEL[] = {"DEBUG", "INFO", "WARN", "FAULT"};
static const char* const MESSAGE[] = {"boot", "range read", "speed set", "range timeout"

int main() {
  RingLog<8> log;
  std::printf("a LogEntry is %zu bytes, a RingLog<8> is %zu bytes\n", sizeof(LogEntry),
  for (int i = 0; i < 12; ++i) {
    uint8_t code = (i == 0) ? 0 : (i == 10) ? 3 : (i == 11) ? 4 : (uint8_t)(1 + i % 2);
    LogLevel lv = (i == 10) ? LOG_WARN : (i == 11) ? LOG_FAULT : (i == 0) ? LOG_INFO : L
    log.add(100u * (uint32_t)i, lv, code, (int16_t)(i * 10));
  }
  std::printf("kept %u, dropped %u\n", (unsigned)log.size(), (unsigned)log.dropped());
  for (uint8_t i = 0; i < log.size(); ++i) {
    const LogEntry& e = log.at(i);
    std::printf("%6u ms  %-5s  %-14s  [%d]\n", (unsigned)e.t, LEVEL[e.level], MESSAGE[e.
  }
  return 0;
}
```

```
a LogEntry is 8 bytes, a RingLog<8> is 72 bytes
kept 8, dropped 4
   400 ms  DEBUG  range read     [40]
   500 ms  DEBUG  speed set      [50]
   600 ms  DEBUG  range read     [60]
```

```
   700 ms  DEBUG  speed set       [70]
   800 ms  DEBUG  range read      [80]
   900 ms  DEBUG  speed set       [90]
  1000 ms  WARN   range timeout   [100]
  1100 ms  FAULT  safe stop       [110]
```

Four events were overwritten and counted (`dropped` is 4), and the dump shows the last eight, ending with the fault and the safe stop. The log reports its own loss: a log that silently loses events cannot be trusted. On the PC `RingLog<8>` is 72 bytes: 8 records of 8 bytes plus two counters and padding.

> **A REAL MISTAKE THE COMPILER HID**
>
> The capacity `N` is a `uint8_t`, so it can be at most 255. I tried a log of 256 records for the Uno, and **it compiled**: "Global variables use 329 bytes", *less* than the 473 bytes of a 16-record log. 256 had been silently truncated to 0, so the log had shrunk to nothing, and the only hint was a `-Woverflow` warning that `arduino-cli` does not show by default (it appeared with `--warnings all`). **What went wrong:** a size too big for its type is converted without a word. **How to avoid it:** make the compiler refuse impossible values; the header now has `static_assert(N > 0, ...)`, and the same build now stops with *static assertion failed: RingLog capacity must be 1..255*.

11. The sketch, and the memory budget

The sketch ties it together: the simulated world advances by the real time that passed, the scheduler runs the three robot tasks at 20 ms and a report task every 500 ms that writes to the log and to `Serial`.

```
// robot_ch1.ino -- Programming for Robotics with Arduino C++, chapter 1: a wall-stoppin
// Compile-checked for the Arduino Uno R3 (arduino:avr:uno). NOT run on a board: no boar
#include "scheduler.h"
#include "log.h"
#include "robot.h"

// These checks cost nothing at run time: if the board's types are not what the chapter
static_assert(sizeof(int) == 2, "int is 16 bits on an Uno");
static_assert(sizeof(void*) == 2, "a pointer is 16 bits on an Uno");
static_assert(sizeof(long) == 4, "long is 32 bits");
static_assert(sizeof(float) == 4, "float is 32 bits");
static_assert(sizeof(double) == 4, "double is the SAME as float on AVR");
```

```
static_assert(sizeof(long long) == 8, "long long is 64 bits");
static_assert(sizeof(LogEntry) == 8, "a log record is 8 bytes");

#ifndef LOGN
#define LOGN 16                                    // log capacity in records; chang
#endif
#ifndef VIRTUAL
#define VIRTUAL 0                                  // 1 = the hardware is reached th
#endif

#if VIRTUAL
struct IHW {                                       // an interface: every call goes
  virtual float range_m() = 0;
  virtual void drive(float v) = 0;
};
struct VSim : IHW {
  SimHW w;
  float range_m() override { return w.range_m(); }
  void drive(float v) override { w.drive(v); }
};
static VSim hw;
static SimHW& world = hw.w;
typedef WallRobot<IHW> Robot;
#else
static SimHW hw(1.0f);
static SimHW& world = hw;
typedef WallRobot<SimHW> Robot;
#endif

static Robot robot(hw, 0.25f, 1.5f, 0.5f);
static Scheduler<4> sch;
static RingLog<LOGN> blackbox;
static uint32_t last_ms = 0;

static void report_task(void*, uint32_t now) {
  blackbox.add(now, LOG_DEBUG, 1, (int16_t)(world.range_m() * 1000.0f));   // range in m
  Serial.print(now);
  Serial.print(F(" ms  range "));
  Serial.print(world.range_m(), 3);
  Serial.println(F(" m"));
}

void setup() {
  Serial.begin(115200);
  sch.add(Robot::sense_task, &robot, 20);
```

```
  sch.add(Robot::think_task, &robot, 20);
  sch.add(Robot::act_task, &robot, 20);
  sch.add(report_task, 0, 500);
  last_ms = millis();
}

void loop() {
  uint32_t now = millis();
  world.advance((now - last_ms) / 1000.0f);           // the simulated world moves by t
  last_ms = now;
  sch.run(now);
}
```

The compiler's report for the Uno, for different log sizes (`-DLOGN=...`), and for the virtual variant of section 8:

BUILD	FLASH USED (OF 32,256)	RAM USED BY GLOBALS (OF 2,048)	LEFT FOR THE STACK
16 records	4950 bytes (15 %)	473 bytes (23 %)	1575 bytes
64 records	4950 bytes	857 bytes (41 %)	1191 bytes
128 records	4950 bytes	1369 bytes (66 %)	679 bytes
192 records	5034 bytes	1881 bytes (91 %)	167 bytes
200 records	5034 bytes	1945 bytes (94 %)	103 bytes
256 records	refused by the `static_assert` (the earlier build was a log of 0 records)		

One record costs exactly 8 bytes (857 - 473 over 48 more records) and everything else in the program costs 345. The “left for the stack” column is **2048 minus the globals**, which is what is *available*; the stack's real depth was not measured, because the sketch was not run. RAM is the scarce thing: a program with a 64-record log already uses 41 % of the RAM, and the 200-record one leaves a hundred bytes for every function call, local variable and interrupt, with no warning from the compiler if it runs out. Flash is not scarce: the whole program is 15 %.

Every chapter of this course ends with such a table: the program's **RAM**, **flash** and **time** budget, from the tools, not from guesses. Time is the one this chapter can only model.

12. Running it on the real board

None of this was run on a board. To do it: connect the Uno, find its port with `arduino-cli board list`, then `arduino-cli compile --fqbn arduino:avr:uno robot_ch1` and `arduino-cli upload -p COM3 --fqbn arduino:avr:uno robot_ch1` (replace `COM3` with the port found), and `arduino-cli monitor -p COM3 -c baudrate=115200` to watch the report lines. You should see a line every 500 ms with the range falling toward 0.250 m, but that is what the code says it will do, not something that was observed: if the board's lines look different, treat that as a finding to investigate.

Challenges

CHALLENGE 1: HOW MANY START TIMES DOES THE NAIVE TIMER SURVIVE?

Run a 20 ms timer, polled every millisecond for 201 ms, starting 0 to 39 ms before the wrap of the counter, both with `Periodic` and with the naive `now >= next` version. For each start report the shortest and longest gap between runs, and count how many of the 40 starts each timer got wrong. A timer is right only if every gap is exactly 20 ms.

View solution

CHALLENGE 2: BUDGET THE BLACK BOX

The stack needs at least 512 bytes of the Uno's 2048 free. From only two compiler reports (a log of 16 and a log of 64 records), predict the largest log that is allowed, then compile with that size and the sizes either side of it to check the prediction.

View solution

CHALLENGE 3: HOW SLOW CAN THE LOOP BE?

Run the wall robot (stop line 0.25 m, top speed 0.5 m/s, the wall at 1.037 m) at loop periods of 20, 100 and 500 ms and gains of 1.5, 5, 20 and 60 per second. For each pair report how many millimetres it ends *past* the stop line, and look for the number (gain × period) at which the problem starts.

View solution

QUICK REFERENCE

- **Board.** Uno R3: 2048 bytes RAM, 32,256 bytes usable flash, `int` 2 bytes, `double` = `float` = 4 bytes, pointer 2 bytes (compile-time checked).
- **Time.** `millis()` wraps after 49.71 days, `micros()` after 71.58 minutes. Compare with `now - then` (unsigned), never `now >= next`. The naive timer failed for 37 of 40 start offsets, and when it works at all it can be by luck (6 ms before the wrap).
- **Rate.** `work(); delay(20)`: 428 passes in 10 s, intervals 21 to 26 ms. A `Periodic(20)` timer: 500 passes, exactly 20.00 ms.
- **Blocking.** A `delay(500)` blink noticed a button after a mean of 496.1 ms and as much as 979 ms; a 10 ms poll, 3.6 ms and 9.
- **Scheduler.** One 35 ms task in a 20 ms slot made sense run 27 times instead of 100 and made a 100 ms task start up to 30 ms late. Tasks must be short.
- **Virtual functions** cost +52 bytes of flash and +10 of RAM here (compile-time). Templates cost nothing.
- **Log.** 8 bytes a record; 148 records is the largest that leaves 512 bytes for the stack (the check at 149 left 511). A capacity of 256 in a `uint8_t` became 0 without a warning until the `static_assert` was added.
- **Loop speed.** At 20 ms the robot is safe up to a gain of 20/s and goes 1.4 mm past the line at 60/s; at 100 ms it goes 13 mm past from 20/s; at 500 ms it goes 55.5 mm past at 5/s and 213 mm at 20/s.
- **Not run:** the sketch on a board (no board attached); every timing here is a fake clock.

Next: Programming for Robotics with Arduino C++ 2, Talking to Hardware

Programming for Robotics with Arduino C++ 2 (planned; not yet written) leaves the simulated world: it reads real pins, drives motors with PWM, and reads sensors, and it takes the `PinHW` that replaces `SimHW` in this chapter's template, the same robot code running against real hardware.

Programming for Robotics with Arduino C++

Course 2 · Chapter 2 · Talking to Hardware

Programming for Robotics with Arduino C++ 1 ended with a robot that drove toward a wall in a *simulated* world: `SimHW` had two functions, `range_m()` and `drive(speed)`, and everything else in the program was written against them. This chapter writes the real `PinHW` that replaces it. That means turning a speed into a PWM duty, a pulse width into a distance, and a stream of bytes from an I2C chip into numbers with units, and doing it on a board where a careless call can cost half the loop time.

The chapter is about the *seams*: the place where physical quantities become C++ values and back. Each seam is kept in a header of plain C++ with no Arduino calls, so that a PC can test it. Each is then connected to the real pins in one small sketch.

WHAT WAS RUN, AND WHAT WAS NOT

Run on a PC (g++ 16.2, C++11, `-Wall -Wextra -pedantic`, no warnings): every block below that has an output under it. These use a *fake clock*, a *fake ultrasonic sensor* and a *fake MPU-6050* written by me from the register map, so they show that **my code does what the documents say**, not what a real sensor does. Noise levels are assumptions and are labelled.

Compiled for the Uno, not run (arduino-cli 1.6.0, core `arduino:avr` 1.8.8): `robot_ch2.ino` and the small sketches whose sizes are compared. Flash and RAM figures, and the disassembly shown in section 1, are the real compiler's. I also read the installed Arduino core source to confirm the timer, serial-buffer and baud-rate facts quoted.

No board, motor, sensor or IMU was attached. No pin was driven, no PWM frequency measured with an oscilloscope, no sensor read. Facts about the chips (register addresses, scale factors, the 0x68 identity) come from my knowledge of the datasheets and should be checked against the datasheet of the part you buy.

1. A pin is three register bits

Every digital pin of the Uno belongs to a *port* (B, C or D), and each port has three 8-bit registers: `DDRx` (direction: 1 = output), `PORTx` (the output value, or the pull-up for an input) and `PINx` (the input value, read-only). Pin 13 is bit 5 of port B. `pinMode(13, OUTPUT)` sets bit 5 of `DDRB`; `digitalWrite(13, HIGH)` sets bit 5 of `PORTB`. That is all a digital output is.

The Arduino functions take a pin *number* and look the port and bit up in tables while the program runs, and they switch off any PWM on the pin and protect the write from interrupts. That is convenient and has a price. Two sketches that toggle pin 13 forever, one with `digitalWrite` and one with the port register directly, were compiled for the Uno:

```
// toggles pin 13 with digitalWrite
void setup() { pinMode(13, OUTPUT); }
void loop() { digitalWrite(13, HIGH); digitalWrite(13, LOW); }
```

```
// toggles pin 13 (PB5) by writing the port register directly
void setup() { DDRB |= _BV(PB5); }
void loop() { PORTB |= _BV(PB5); PORTB &= ~_BV(PB5); }
```

SKETCH	FLASH	RAM
empty `setup()` and `loop()`	444 bytes	9 bytes
toggle with `digitalWrite`	734 bytes (+290)	9 bytes
toggle with `PORTB`	450 bytes (+6)	9 bytes

The disassembly of the port version's loop, from the compiler's own output (`avr-objdump -d`), is two instructions and the loop jump: `sbi` sets one bit of a register, `cbi` clears one, and the instruction set manual gives each as 2 clock cycles. The specialised copy of `digitalWrite` the compiler made for the other sketch is 144 bytes, 66 instructions, and begins like this:

```
1b0:    2d 9a           sbi     0x05, 5 ; 5
1b2:    2d 98           cbi     0x05, 5 ; 5
1b4:    20 97           sbiw    r28, 0x00       ; 0
1b6:    e1 f3           breq    .-8             ; 0x1b0 <main+0x8c>
```

```
000000e0 <digitalWrite.constprop.0>:
e0: e1 eb       ldi r30, 0xB1 ; 177
e2: f0 e0       ldi r31, 0x00 ; 0
e4: 24 91       lpm r18, Z
e6: ed e9       ldi r30, 0x9D ; 157
e8: f0 e0       ldi r31, 0x00 ; 0
ea: 94 91       lpm r25, Z
ec: e9 e8       ldi r30, 0x89 ; 137
```

```
  ee: f0 e0        ldi r31, 0x00 ; 0
  f0: e4 91        lpm r30, Z
  f2: ee 23        and r30, r30
  f4: c9 f0        breq .+50      ; 0x128 <digitalWrite.constprop.0+0x48>
  f6: 22 23        and r18, r18
  f8: 39 f0        breq .+14      ; 0x108 <digitalWrite.constprop.0+0x28>
  fa: 23 30        cpi r18, 0x03 ; 3
  fc: 01 f1        breq .+64      ; 0x13e <digitalWrite.constprop.0+0x5e>
  fe: a8 f4        brcc .+42      ; 0x12a <digitalWrite.constprop.0+0x4a>
 100: 21 30         cpi r18, 0x01 ; 1
 102: 19 f1         breq .+70      ; 0x14a <digitalWrite.constprop.0+0x6a>
 104: 22 30         cpi r18, 0x02 ; 2
```

The three `lpm` instructions read the pin's port, bit and timer from tables in flash; further down it saves the status register and disables interrupts (`cli`) while it writes. **This chapter does not time the two on a board**, so no speed ratio is claimed: the evidence is size and the visible instructions. The honest rule is the same as in Programming for Robotics with Arduino C++ 1: use `digitalWrite` by default, because it is clear and safe, and go to the register only where the instruction count has been shown to matter (a step pulse at tens of kilohertz, say). The register form is *unsafe* in one way `digitalWrite` is not: it knows nothing about PWM, so writing a bit that a timer is driving leaves the timer in control.

2. A pin table the compiler checks

Not every pin can do every job: only six can do PWM, only two can interrupt, pins 0 and 1 are the USB serial link, and A4 and A5 are the I2C bus. A wiring mistake of this kind compiles perfectly and fails on the bench. Writing the capabilities as a `constexpr` table lets the compiler check the plan:

```
// pins.h -- what each Uno R3 pin can do, and the build robot's pin plan, checked by the
#pragma once
#include <stdint.h>

enum Cap : uint8_t { CAP_DIGITAL = 1, CAP_PWM = 2, CAP_ADC = 4, CAP_INT = 8, CAP_I2C = 1(
enum Role : uint8_t { R_DIGITAL, R_PWM, R_ADC, R_INTERRUPT };

// Pin numbers as the Arduino core uses them: 0..13 digital, 14..19 = A0..A5.
const uint8_t PIN_COUNT = 20;
constexpr uint8_t PIN_CAPS[PIN_COUNT] = {
```

```
  CAP_DIGITAL | CAP_UART,                 //  0  RX  (USB serial)
  CAP_DIGITAL | CAP_UART,                 //  1  TX  (USB serial)
  CAP_DIGITAL | CAP_INT,                  //  2  INT0
  CAP_DIGITAL | CAP_PWM | CAP_INT,        //  3  INT1, Timer2 B
  CAP_DIGITAL,                            //  4
  CAP_DIGITAL | CAP_PWM,                  //  5  Timer0 B
  CAP_DIGITAL | CAP_PWM,                  //  6  Timer0 A
  CAP_DIGITAL,                            //  7
  CAP_DIGITAL,                            //  8
  CAP_DIGITAL | CAP_PWM,                  //  9  Timer1 A
  CAP_DIGITAL | CAP_PWM | CAP_SPI,        // 10  Timer1 B, SS
  CAP_DIGITAL | CAP_PWM | CAP_SPI,        // 11  Timer2 A, MOSI
  CAP_DIGITAL | CAP_SPI,                  // 12  MISO
  CAP_DIGITAL | CAP_SPI,                  // 13  SCK, and the on-board LED
  CAP_DIGITAL | CAP_ADC,                  // 14  A0
  CAP_DIGITAL | CAP_ADC,                  // 15  A1
  CAP_DIGITAL | CAP_ADC,                  // 16  A2
  CAP_DIGITAL | CAP_ADC,                  // 17  A3
  CAP_DIGITAL | CAP_ADC | CAP_I2C,        // 18  A4, SDA
  CAP_DIGITAL | CAP_ADC | CAP_I2C,        // 19  A5, SCL
};

constexpr uint8_t role_cap(Role r) { return r == R_PWM ? CAP_PWM : r == R_ADC ? CAP_ADC

// Pins that a plan may not hand out: 0 and 1 are the USB serial link, 18 and 19 belong
constexpr bool pin_reserved(uint8_t p) { return p == 0 || p == 1 || p == 18 || p == 19;
constexpr bool pin_can(uint8_t p, Role r) { return p < PIN_COUNT && (PIN_CAPS[p] & role_
constexpr bool pin_ok(uint8_t p, Role r) { return pin_can(p, r) && !pin_reserved(p); }

// The build robot's plan. Each line is checked at compile time: a wrong pin stops the bu
const uint8_t PIN_ML_PWM = 5, PIN_ML_DIR = 4;           // left motor: PWM on Timer0 B,
const uint8_t PIN_MR_PWM = 6, PIN_MR_DIR = 7;           // right motor: PWM on Timer0 A
const uint8_t PIN_TRIG = 8, PIN_ECHO = 12;              // ultrasonic sensor
const uint8_t PIN_ENC_L = 2, PIN_ENC_R = 3;             // wheel encoders: the only two
const uint8_t PIN_BUMP = 11;                            // bumper switch
const uint8_t PIN_BATT = 14;                            // A0: battery voltage through
const uint8_t PIN_LED = 13;

static_assert(pin_ok(PIN_ML_PWM, R_PWM) && pin_ok(PIN_MR_PWM, R_PWM), "a motor speed pin
static_assert(pin_ok(PIN_ML_DIR, R_DIGITAL) && pin_ok(PIN_MR_DIR, R_DIGITAL), "direction
static_assert(pin_ok(PIN_TRIG, R_DIGITAL) && pin_ok(PIN_ECHO, R_DIGITAL), "sensor pins mu
static_assert(pin_ok(PIN_ENC_L, R_INTERRUPT) && pin_ok(PIN_ENC_R, R_INTERRUPT), "encoders
static_assert(pin_ok(PIN_BUMP, R_DIGITAL) && pin_ok(PIN_LED, R_DIGITAL), "switch and LED
static_assert(pin_ok(PIN_BATT, R_ADC), "the battery monitor needs an analog input that I
```

The same checks, run on a PC against the plan and against six plans with one mistake each:

```
// e_pins.cpp -- the pin plan through the same checks the compiler uses, then a series o
#include <cstdio>
#include "pins.h"

struct Use { const char* name; uint8_t pin; Role role; };
static const char* ROLE_NAME[] = {"digital", "PWM", "analog in", "interrupt"};

// Returns the number of problems; prints each one.
static int check(const Use* u, int n) {
  int bad = 0;
  for (int i = 0; i < n; ++i) {
    if (!pin_can(u[i].pin, u[i].role)) { std::printf("    ERROR  %-14s pin %2u cannot do
    else if (pin_reserved(u[i].pin)) { std::printf("    ERROR  %-14s pin %2u is reserved
    for (int j = 0; j < i; ++j)
      if (u[j].pin == u[i].pin) { std::printf("    ERROR  %-14s pin %2u is already used
  }
  return bad;
}

int main() {
  Use plan[] = {
    {"left PWM", PIN_ML_PWM, R_PWM},   {"left dir", PIN_ML_DIR, R_DIGITAL},
    {"right PWM", PIN_MR_PWM, R_PWM},  {"right dir", PIN_MR_DIR, R_DIGITAL},
    {"trigger", PIN_TRIG, R_DIGITAL},  {"echo", PIN_ECHO, R_DIGITAL},
    {"encoder L", PIN_ENC_L, R_INTERRUPT}, {"encoder R", PIN_ENC_R, R_INTERRUPT},
    {"bumper", PIN_BUMP, R_DIGITAL},   {"battery", PIN_BATT, R_ADC}, {"LED", PIN_LED, R_
  };
  const int n = (int)(sizeof plan / sizeof plan[0]);
  std::printf("the build robot's plan (%d pins):\n", n);
  for (int i = 0; i < n; ++i) std::printf("  %-10s pin %2u  %s\n", plan[i].name, (unsign
  std::printf("  problems: %d\n", check(plan, n));

  struct Mistake { const char* what; int index; uint8_t pin; };
  const Mistake mistakes[] = {
    {"left motor PWM moved to pin 4", 0, 4},
    {"left encoder moved to pin 4", 6, 4},
    {"echo wired to pin 1", 5, 1},
    {"battery monitor moved to pin 7", 9, 7},
    {"bumper on the same pin as the left direction line", 8, 4},
    {"battery monitor on A4 (an I2C pin)", 9, 18},
  };
  std::printf("\nsix plans, each with one change:\n");
```

```
  for (const Mistake& m : mistakes) {
    Use copy[11];
    for (int i = 0; i < n; ++i) copy[i] = plan[i];
    copy[m.index].pin = m.pin;
    std::printf("  %s:\n", m.what);
    int bad = check(copy, n);
    if (bad == 0) std::printf("    (no problem found)\n");
  }
  return 0;
}
```

```
the build robot's plan (11 pins):
  left PWM   pin  5  PWM
  left dir   pin  4  digital
  right PWM  pin  6  PWM
  right dir  pin  7  digital
  trigger    pin  8  digital
  echo       pin 12  digital
  encoder L  pin  2  interrupt
  encoder R  pin  3  interrupt
  bumper     pin 11  digital
  battery    pin 14  analog in
  LED        pin 13  digital
  problems: 0

six plans, each with one change:
  left motor PWM moved to pin 4:
    ERROR  left PWM       pin  4 cannot do PWM
    ERROR  left dir       pin  4 is already used by left PWM
  left encoder moved to pin 4:
    ERROR  encoder L      pin  4 cannot do interrupt
    ERROR  encoder L      pin  4 is already used by left dir
  echo wired to pin 1:
    ERROR  echo           pin  1 is reserved (USB serial or I2C)
  battery monitor moved to pin 7:
    ERROR  battery        pin  7 cannot do analog in
    ERROR  battery        pin  7 is already used by right dir
  bumper on the same pin as the left direction line:
    ERROR  bumper         pin  4 is already used by left dir
  battery monitor on A4 (an I2C pin):
    ERROR  battery        pin 18 is reserved (USB serial or I2C)
```

And for the board, the same header makes a wrong pin a *build* error. With the left motor's speed pin moved to pin 4 (and its direction line to 9), the Uno build stopped with:

```
static assertion failed: a motor speed pin must be PWM-capable and not reserved
```

The table is my transcription of the Uno R3 pin map. If a pin is wrong in the table, the check is wrong in the same way, so it is worth verifying against the board's own pinout once.

3. PWM: three timers, and one of them keeps the time

`analogWrite(pin, 0..255)` does not produce an analog voltage. It switches the pin fully on and fully off at a fixed frequency, and the fraction of each period spent on is the duty. A motor, which responds slowly, sees the average. The pulses come from one of the chip's three **timers**, each of which drives two pins: Timer0 pins 5 and 6, Timer1 pins 9 and 10, Timer2 pins 3 and 11. The frequency is the clock divided by the prescaler times the number of steps, 256 in fast mode and 510 in phase-correct mode (which counts up and down):

```
// e_pwm.cpp -- PWM frequencies of the Uno's three timers, and what an 8-bit duty does t
#include <cstdio>
#include <cmath>
#include "conv.h"

int main() {
  const double F = 16000000.0;
  std::printf("PWM frequency = F_CPU / (prescaler x steps); fast PWM has 256 steps, phas
  std::printf("  timer   mode           prescaler   frequency Hz   period us   pins\n");
  struct Row { const char* timer; const char* mode; int pre; int steps; const char* pins;
  const Row rows[] = {
    {"Timer0", "fast (Arduino)", 64, 256, "5, 6"},
    {"Timer1", "phase-correct", 64, 510, "9, 10"},
    {"Timer2", "phase-correct", 64, 510, "3, 11"},
    {"Timer1", "phase-correct", 8, 510, "9, 10"},
    {"Timer1", "phase-correct", 1, 510, "9, 10"},
    {"Timer1", "fast", 1, 256, "9, 10"},
  };
  for (const Row& r : rows) {
    double hz = F / ((double)r.pre * r.steps);
    std::printf("  %-6s  %-14s  %9d   %12.2f   %9.2f   %s\n", r.timer, r.mode, r.pre, hz,
  }
```

```
  std::printf("\n8-bit duty: speed 0..0.500 m/s becomes a count 0..255, one count = %.3f
  double worst_round = 0, worst_trunc = 0, sum_trunc = 0;
  int n = 0;
  for (int i = 0; i <= 500; ++i) {                          // every commanded speed in
    float v = i / 1000.0f;
    double er = std::fabs(speed_to_pwm(v, 0.5f) * 0.5 / 255.0 - v) * 1000.0;
    double et = std::fabs(speed_to_pwm_truncating(v, 0.5f) * 0.5 / 255.0 - v) * 1000.0;
    if (er > worst_round) worst_round = er;
    if (et > worst_trunc) worst_trunc = et;
    sum_trunc += (speed_to_pwm_truncating(v, 0.5f) * 0.5 / 255.0 - v) * 1000.0;
    ++n;
  }
  std::printf("  worst speed error with rounding:    %.3f mm/s\n", worst_round);
  std::printf("  worst speed error with truncation:  %.3f mm/s (mean signed error %.3f m
  std::printf("  speed_to_pwm(0.0009) = %u, (0.0010) = %u, (0.2500) = %u, (0.5000) = %u,
              (unsigned)speed_to_pwm(0.0009f, 0.5f), (unsigned)speed_to_pwm(0.0010f, 0.5
              (unsigned)speed_to_pwm(0.5f, 0.5f), (unsigned)speed_to_pwm(0.9f, 0.5f), (u
  return 0;
}
```

```
PWM frequency = F_CPU / (prescaler x steps); fast PWM has 256 steps, phase-correct 510:
  timer   mode            prescaler   frequency Hz   period us   pins
  Timer0  fast (Arduino)         64         976.56     1024.00   5, 6
  Timer1  phase-correct          64         490.20     2040.00   9, 10
  Timer2  phase-correct          64         490.20     2040.00   3, 11
  Timer1  phase-correct           8        3921.57      255.00   9, 10
  Timer1  phase-correct           1       31372.55       31.88   9, 10
  Timer1  fast                    1       62500.00       16.00   9, 10

8-bit duty: speed 0..0.500 m/s becomes a count 0..255, one count = 1.961 mm/s
  worst speed error with rounding:    0.980 mm/s
  worst speed error with truncation:  1.941 mm/s (mean signed error -0.969 mm/s: always
  speed_to_pwm(0.0009) = 0, (0.0010) = 1, (0.2500) = 128, (0.5000) = 255, (0.9000) = 255
```

Two settings in the table matter for a robot. First, **the defaults are low frequencies**: 490 Hz and 977 Hz are within hearing, so a motor driven by them typically whines, and a period of about 2 ms is long enough that a small motor may run unevenly (not tested here). Setting a timer to prescaler 1 gives 31 kHz, beyond hearing. Second, **Timer0 is also the clock.** The Arduino core counts Timer0's overflows to produce `millis()` and `micros()` (it is in the core's `wiring.c`, which I read: the overflow period is hard-coded as 64 × 256 clock cycles). If a program changes Timer0's prescaler to quiet motors on pins 5 and 6, every time in the program, and every timer

built on it from Programming for Robotics with Arduino C++ 1, runs at the wrong rate. The build robot's two motors are on Timer0 (pins 5 and 6) *because* that is simple, and so the plan leaves Timer0 alone; a quieter drive would put the motors on Timer2's pins (3 and 11) and then give up pin 3 for an interrupt. A plan, not a default, decides.

The second half of the output is the other seam, the speed to duty conversion. 255 steps over 0.5 m/s is 1.961 mm/s a step, so the controller can only ask for speeds on that grid. `speed_to_pwm` adds 0.5 before the cast to round to the nearest step; without it the C++ cast truncates, and the speed is *always* a little low (mean error -0.969 mm/s, worst 1.941, against 0.980 for rounding). The same function clamps, so a command of 0.9 m/s gives 255 and -0.1 gives 0, never a value that wraps around in a `uint8_t`. **It does not say 0.5 m/s is what a duty of 255 will really give**: that depends on the motors, the battery and the load, and has to be measured on the robot (a calibration the chapter does not have, which is why `V_MAX_MPS` is labelled an assumption in the sketch).

4. The ADC: ten bits and 104 microseconds

`analogRead(pin)` returns 0 to 1023 for 0 to 5 V (the reference is the supply, so a supply that sags makes every reading wrong in step). One count is 4.888 mV. A conversion takes 13 ADC clocks, and the Arduino core sets the ADC clock to 125 kHz, so **one reading takes 104 microseconds** and `analogRead` waits for it. Noise on the input shows up as a spread in the counts; averaging N readings reduces the spread by $\sqrt{N}$, at N times the time:

```
// e_adc.cpp -- the 10-bit ADC: step size, conversion time, and averaging noisy readings
#include <cstdio>
#include <cmath>
#include <random>
#include "conv.h"

int main() {
  const double ADC_CLOCK_HZ = 16000000.0 / 128.0;          // Arduino sets the prescale
  const double conv_us = 13.0 / ADC_CLOCK_HZ * 1.0e6;      // one conversion takes 13 A
  std::printf("one count = %.3f mV (5 V / 1023); ADC clock %.0f Hz; one conversion = %.0
              5000.0 / 1023.0, ADC_CLOCK_HZ, conv_us, 1.0e6 / conv_us);

  double worst = 0;
  for (uint16_t c = 0; c <= 1023; ++c) {
    double d = std::fabs(counts_to_mv(c) - counts_to_volts(c) * 1000.0);
    if (d > worst) worst = d;
```

```
  }
  std::printf("integer counts_to_mv() against the float version, all 1024 counts: worst d

  std::mt19937 rng(11);
  std::normal_distribution<double> noise(0.0, 1.5);         // ASSUMED: 1.5 counts of noi
  const double true_counts = 409.6;                          // 2.000 V: between two coun
  std::printf("true input 2.000 V = 409.6 counts, noise sigma 1.5 counts (assumed), 4000
  std::printf("  readings averaged   spread (std dev, counts)   spread mV   time per answ
  for (int N : {1, 4, 16, 64, 256}) {
    double sum = 0, sum2 = 0;
    const int trials = 4000;
    for (int t = 0; t < trials; ++t) {
      long total = 0;
      for (int i = 0; i < N; ++i) {
        double x = true_counts + noise(rng);
        long c = std::lround(x);                             // the converter rounds to a
        if (c < 0) c = 0;
        if (c > 1023) c = 1023;
        total += c;
      }
      double avg = (double)total / N;
      sum += avg;
      sum2 += avg * avg;
    }
    double mean = sum / trials, sd = std::sqrt(sum2 / trials - mean * mean);
    std::printf("  %17d   %26.3f   %9.2f   %18.0f   %16.3f\n", N, sd, sd * 5000.0 / 1023
  }
  // The caveat: averaging only helps when noise dithers the reading across counts. With
  long same = std::lround(true_counts);
  std::printf("\nwith no noise at all, 256 readings of 409.6 counts average to %ld: %.1f
              same, std::fabs(same - true_counts), std::fabs(same - true_counts) * 5000.0
  return 0;
}
```

```
one count = 4.888 mV (5 V / 1023); ADC clock 125000 Hz; one conversion = 104 us = 9615 r
integer counts_to_mv() against the float version, all 1024 counts: worst difference 0.49

true input 2.000 V = 409.6 counts, noise sigma 1.5 counts (assumed), 4000 trials per row
  readings averaged   spread (std dev, counts)   spread mV   time per answer us   spread
                  1                       1.547        7.56                  104
                  4                       0.764        3.74                  416
                 16                       0.381        1.86                 1664
                 64                       0.190        0.93                 6656
                256                       0.095        0.46                26624
```

```
with no noise at all, 256 readings of 409.6 counts average to 410: 0.4 counts (1.96 mV)
```

The last column is nearly constant (1.52): the spread falls as 1/√N, and 64 readings take a 1.5-count noise down to 0.93 mV in 6.66 ms. **The noise is an assumption** of 1.5 counts, a moderate input, and the figures follow from that, not from a measured input. The last line is the caveat: if there is no noise at all, every reading is the same whole count and averaging cannot recover the 0.4 count between them. Averaging helps only because noise spreads the readings across neighbouring counts.

Converting counts to a voltage can be done in integers: `counts_to_mv` multiplies by 5000 in a `uint32_t` and divides by 1023, rounding, and it differs from the float version by at most 0.499 mV. Compiled for the Uno, a sketch that does this and writes a PWM duty with the *integer* version is 1118 bytes of flash; the identical sketch with the *float* version is 1818 bytes, **700 bytes larger**, because the AVR has no floating-point hardware and the float multiply and divide come from a software library. The robot code still uses floats (the controller needs them) and the cost is paid once; the lesson is not to add a second float function for a job that integers do exactly.

5. A sensor in digital pins, without blocking

The common HC-SR04 ultrasonic sensor has two pins: a 10-microsecond pulse on *trigger* sends a burst of sound, and *echo* then goes high for the time the sound takes to travel to an object and back. The distance is the width of that pulse times the speed of sound (343 m/s in dry air at about 20°C) over two. Arduino's `pulseIn()` measures the width, but it **waits inside the call** until the pulse ends or its timeout passes, which is exactly the `delay()` problem of Programming for Robotics with Arduino C++ 1 in a new coat. How long it would block a 20 ms control period:

```
// e_echo.cpp -- the non-blocking echo driver against a fake sensor, polled every 10, 10
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "echo.h"
#include "conv.h"

struct FakeSensor {                                    // an object at distance d:
  float d_m;
```

```
  uint32_t t_trig_fall;
  bool trig_prev;
  bool armed;
  explicit FakeSensor(float d) : d_m(d), t_trig_fall(0), trig_prev(false), armed(false) 
  bool echo(uint32_t now, bool trig) {
    if (trig_prev && !trig) { t_trig_fall = now; armed = true; }   // the trigger just f
    trig_prev = trig;
    if (!armed || d_m > 4.0f) return false;                  // beyond about 4 m there i
    uint32_t w = m_to_echo_us(d_m);
    return now >= t_trig_fall && now < t_trig_fall + w;
  }
};

// Returns the measured distance in metres, or -1 on a timeout. `waited_us` is the time 
static float measure(float d, uint32_t poll_us, uint32_t* took_us) {
  Echo e;
  FakeSensor s(d);
  bool trig = false;
  uint32_t t = 1000;                                         // an arbitrary start time
  e.start();
  uint32_t t_start = t;
  for (;;) {
    bool echo = s.echo(t, trig);
    Echo::Status st = e.update(t, echo, trig);
    if (st != Echo::BUSY) {
      *took_us = t - t_start;
      return st == Echo::READY ? echo_us_to_m(e.width_us()) : -1.0f;
    }
    t += poll_us;
  }
}

int main() {
  std::printf("worst distance error of the polled state machine over distances 0.10 to 3
  std::printf("  polled every   worst error mm   expected: one poll = %s\n", "poll_us x 
  for (uint32_t poll : {10u, 100u, 1000u}) {
    double worst = 0;
    for (int i = 0; i <= 66; ++i) {
      float d = 0.10f + 0.05f * i;
      uint32_t took;
      float m = measure(d, poll, &took);
      double err = std::fabs(m - d) * 1000.0;
      if (err > worst) worst = err;
    }
    std::printf("  %9u us   %13.1f   %.1f mm\n", (unsigned)poll, worst, poll * 0.1715);
```

```
  }

  std::printf("\ntime the board is stuck inside pulseIn() for one reading, against a 20 m
  std::printf("  object at   echo width us   blocked ms   share of a 20 ms period\n");
  for (float d : {0.3f, 1.0f, 2.0f, 3.5f}) {
    uint32_t w = m_to_echo_us(d);
    std::printf("  %6.1f m    %12u   %10.2f   %6.1f %%\n", d, (unsigned)w, (w + 10) / 10
  }
  std::printf("  nothing there: pulseIn() waits its timeout, 30 ms by the setting used h
              30.0, 30.0 / 20.0 * 100.0);

  uint32_t took;
  float m = measure(2.0f, 100, &took);
  std::printf("\nnon-blocking, object at 2.000 m, polled every 100 us: measured %.3f m, t
  m = measure(5.0f, 100, &took);
  std::printf("object at 5.000 m (out of range): result %s after %u us\n", m < 0 ? "TIME
  return 0;
}
```

```
worst distance error of the polled state machine over distances 0.10 to 3.50 m in 50 mm s
  polled every   worst error mm   expected: one poll = poll_us x 0.1715 mm/us
         10 us             1.6   1.7 mm
        100 us            17.0   17.2 mm
       1000 us           165.5   171.5 mm

time the board is stuck inside pulseIn() for one reading, against a 20 ms control period
  object at   echo width us   blocked ms   share of a 20 ms period
     0.3 m            1749         1.76      8.8 %
     1.0 m            5831         5.84     29.2 %
     2.0 m           11662        11.67     58.4 %
     3.5 m           20408        20.42    102.1 %
  nothing there: pulseIn() waits its timeout, 30 ms by the setting used here: 30.0 ms, 15

non-blocking, object at 2.000 m, polled every 100 us: measured 2.007 m, took 11900 us of
object at 5.000 m (out of range): result TIMEOUT (-1) after 30100 us
```

At 1 m `pulseIn` holds the board for 5.84 ms (29 % of a 20 ms period); at 3.5 m it holds it for the whole period (102 %); and with nothing in front of the sensor it waits for its full timeout, longer than the period. A robot that is stuck facing open space would run its control loop late exactly when it has nothing to see. The driver in `echo.h` is therefore a **state machine**: *start* makes it raise the trigger, *update* is called as often as the loop can, it is handed the time and the echo

pin's level and gives back the trigger's level, and it measures the width itself from the moment it sees the pin rise to the moment it sees it fall.

```
// echo.h -- a NON-BLOCKING driver for an HC-SR04 style ultrasonic sensor, as a state ma
// the caller passes in the time and the echo level and gets back the trigger level to w
#pragma once
#include <stdint.h>
#include "timing.h"

class Echo {
 public:
  enum Status : uint8_t { BUSY, READY, TIMEOUT };
  enum State : uint8_t { IDLE, TRIG_HIGH, WAIT_RISE, WAIT_FALL };

  Echo() : state_(IDLE), t0_(0), rise_(0), width_(0) {}

  void start() { if (state_ == IDLE) state_ = TRIG_HIGH; }   // ask for one measurement;
  bool idle() const { return state_ == IDLE; }
  uint32_t width_us() const { return width_; }               // valid after READY

  // Call as often as you like. `trig` is the level the trigger pin should have after th
  Status update(uint32_t now_us, bool echo, bool& trig) {
    switch (state_) {
      case IDLE:
        trig = false;
        return BUSY;
      case TRIG_HIGH:                                        // the sensor wants a trig
        if (t0_ == 0 && !trig) { trig = true; t0_ = now_us ? now_us : 1; return BUSY; }
        if (elapsed(now_us, t0_) >= 10) { trig = false; state_ = WAIT_RISE; }
        return BUSY;
      case WAIT_RISE:
        if (echo) { rise_ = now_us; state_ = WAIT_FALL; return BUSY; }
        if (elapsed(now_us, t0_) > TIMEOUT_US) return finish(TIMEOUT);
        return BUSY;
      case WAIT_FALL:
        if (!echo) { width_ = elapsed(now_us, rise_); return finish(READY); }
        if (elapsed(now_us, t0_) > TIMEOUT_US) return finish(TIMEOUT);
        return BUSY;
    }
    return BUSY;
  }

  static const uint32_t TIMEOUT_US = 30000;                  // 30 ms after the trigger
```

```
 private:
  Status finish(Status s) { state_ = IDLE; t0_ = 0; return s; }
  State state_;
  uint32_t t0_, rise_, width_;
};
```

The first block of the output shows the price. The width is measured by *polling*, so the error is up to one poll interval: 17.0 mm when polled every 100 microseconds, 165.5 mm when polled every millisecond, in line with the 0.1715 mm of distance in each microsecond (the speed of sound halved). The program was free for all of the measurement except the polls. **This is a fake sensor**: it models the echo as a clean pulse starting the instant the trigger falls, which ignores the sensor's burst delay and any noise, and it never lets two readings overlap. A real sensor adds a fixed offset and variation, and the machine's lower limit on what it can resolve then depends on how fast the real `loop()` runs, which was not measured. Challenge 1 asks how fast that has to be for 10 mm, and a pin-change interrupt can replace the polling (the interrupt tools arrive in Programming for Robotics with Arduino C++ 4).

6. UART, I2C and SPI, and what each costs

Three buses carry most of what a robot talks to, and the first thing to know about each is its time cost. They are compared by arithmetic here, and then by the memory their libraries use.

THE THREE BUSES IN ONE PARAGRAPH

UART (`Serial`): two wires, no clock, both ends must agree on the rate; one byte is 10 bits on the wire (start, 8 data, stop). **I2C** (`Wire`): two wires shared by many devices, each with an address; every byte is followed by an acknowledge bit (9 clocks a byte); the standard clocks are 100 and 400 kHz. **SPI**: a clock, data out, data in and one select line per device; no addresses and no acknowledge, and fast (a clock of the CPU clock divided by 2 to 128).

```
// e_bus.cpp -- what UART, I2C and SPI cost in time: baud-rate error, how long a log lin
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "conv.h"

int main() {
```

```
  const uint32_t F = 16000000UL;
  std::printf("UART: the divisor UBRR gives the nearest rate the clock can make (U2X = d
  std::printf("  wanted baud   mode     UBRR   actual baud   error %%\n");
  for (uint32_t baud : {9600UL, 57600UL, 115200UL, 230400UL, 250000UL, 500000UL, 1000000
    uint16_t un = uart_ubrr_normal(F, baud), u2 = uart_ubrr_u2x(F, baud);
    double an = F / (16.0 * (un + 1)), a2 = F / (8.0 * (u2 + 1));
    std::printf("  %11lu   normal  %5u   %11.0f   %+7.2f\n", (unsigned long)baud, (unsig
    std::printf("  %11s   U2X     %5u   %11.0f   %+7.2f\n", "", (unsigned)u2, a2, (a2 - 
  }

  std::printf("\na 40-character log line (10 bits per character on the wire) against a 2
  std::printf("  baud      us per character   time for 40 characters ms   fits in 20 ms?
  for (uint32_t baud : {9600UL, 57600UL, 115200UL, 250000UL}) {
    double us_char = 10.0 / baud * 1.0e6, ms_line = 40.0 * us_char / 1000.0;
    std::printf("  %6lu    %15.1f   %25.2f   %s\n", (unsigned long)baud, us_char, ms_lin
  }

  std::printf("\nreading 14 bytes from a sensor register (one register-address byte firs
  std::printf("  bus clock     bus clocks   time us   share of a 20 ms period\n");
  for (uint32_t scl : {100000UL, 400000UL}) {
    uint32_t us = i2c_transfer_us(scl, 14, 1);
    std::printf("  I2C %6lu    %9u   %7lu   %6.2f %%\n", (unsigned long)scl, (unsigned)(
  }
  for (uint8_t div : {4, 16, 128}) {
    uint32_t us = spi_transfer_us(F, div, 15);              // a register address byte p
    std::printf("  SPI F/%-3u     %9u   %7lu   %6.2f %%\n", (unsigned)div, 8 * 15, (unsi
  }
  return 0;
}
```

```
UART: the divisor UBRR gives the nearest rate the clock can make (U2X = double-speed mod
  wanted baud   mode     UBRR   actual baud   error %
         9600   normal    103          9615     +0.16
                U2X       207          9615     +0.16
        57600   normal     16         58824     +2.12
                U2X        34         57143     -0.79
       115200   normal      8        111111     -3.55
                U2X        16        125000     +8.51
       230400   normal      3        250000     +8.51
                U2X         8        222222     -3.55
       250000   normal      3        250000     +0.00
                U2X         7        250000     +0.00
       500000   normal      1        500000     +0.00
                U2X         3        500000     +0.00
```

```
       1000000   normal      0       1000000     +0.00
                 U2X         1       1000000     +0.00

  a 40-character log line (10 bits per character on the wire) against a 20 ms control peri
    baud      us per character   time for 40 characters ms   fits in 20 ms?
      9600             1041.7                       41.67   NO: the 64-byte buffer fills a
     57600              173.6                        6.94   yes
    115200               86.8                        3.47   yes
    250000               40.0                        1.60   yes

  reading 14 bytes from a sensor register (one register-address byte first), by bus clock:
    bus clock     bus clocks   time us   share of a 20 ms period
    I2C 100000           146      1460     7.30 %
    I2C 400000           146       365     1.82 %
    SPI F/4              120        30     0.15 %
    SPI F/16             120       120     0.60 %
    SPI F/128            120       960     4.80 %
```

Three results to remember. **Baud rates are approximations**: 115200 is not available exactly, and the divider gives 111111 (-3.55 %) in normal mode or 117647 (+2.12 %) in double-speed mode. The Arduino core uses double-speed mode (I read it: it computes the divisor for double speed first, and falls back to normal only when the divisor does not fit, and for 57600 on a 16 MHz board as a special case), so Serial at 115200 runs 2.12 % fast, which works against a PC whose own rate is accurate because UART receivers tolerate a few percent. At 230400 the normal-mode error is 8.51 % and would not.

Printing is a transfer, and the bytes leave at one every 86.8 microseconds at 115200 (10 bits). A 40-character line takes 3.47 ms; at 9600 baud it takes 41.67 ms, longer than the loop. The Uno's `Serial` has a 64-byte transmit buffer, and when it is full `print()` **waits** (the core's `write()` spins on a full buffer; I read it). That is the same trap as `delay()`: challenge 3 puts a number on it.

An I2C register read has a fixed cost: 14 bytes plus the register address is 146 bus clocks, 1460 microseconds at 100 kHz (7.3 % of a 20 ms period) and 365 at 400 kHz (1.82 %). The cost is the same whether the device is fast or slow, and six devices, each read for 14 bytes every period at 100 kHz, would spend over 40 % of every period on the bus. (The 146 counts 9 clocks a byte for the address byte, the register byte and the 14 data bytes, plus two for the start and stop; it ignores the repeated start and any time the device stretches the clock, so a real transfer is a little longer.) The library costs are real compiler reports:

SKETCH (EACH DOES ONE TRANSFER IN `loop()`)	FLASH	RAM
empty	444 bytes	9 bytes
+ `Serial` at 115200, one `println(F(...))`	1492 bytes (+1048)	188 bytes (+179)
+ `Wire` (I2C) at 400 kHz	2876 bytes (+2432)	225 bytes (+216)
+ `SPI`	840 bytes (+396)	10 bytes (+1)

`Wire` costs 216 bytes of RAM, over 10 % of the board (the library keeps send and receive buffers; the compiler's report does not break the figure down); `Serial` costs 179 bytes for its 64-byte transmit buffer and 64-byte receive buffer and bookkeeping; SPI costs almost nothing. These are the first of the bills that the budget table at the end of the chapter adds up.

7. Reading a datasheet: a register map into a driver

An I2C chip is a numbered list of one-byte registers. A datasheet's *register map* says what each means; a driver is a translation of that table into code. The example is the MPU-6050, a common six-axis motion sensor, whose map (as I know it; **check the datasheet of your part**) has these entries that matter here:

REGISTER	ADDRESS	WHAT THE MAP SAYS
`WHO_AM_I`	0x75	always reads 0x68
`PWR_MGMT_1`	0x6B	bit 6 is SLEEP and is **set** at power-up: the chip starts asleep
`ACCEL_CONFIG` / `GYRO_CONFIG`	0x1C / 0x1B	bits 4:3 pick the range: ±2/4/8/16 g (16384/8192/4096/2048 counts per g) and ±250/500/1000/2000 deg/s (131/65.5/32.8/16.4 counts per deg/s)
`ACCEL_XOUT_H` …	0x3B … 0x48	14 bytes: accelerometer X, Y, Z, temperature, gyro X, Y, Z; each a signed 16-bit number, high byte first; temperature in °C is raw / 340 + 36.53

The driver is a template on the bus, the same idea as the robot being a template on its hardware, so the PC can test it with a fake chip and the board runs it over `Wire`. It reads all 14 bytes in **one burst**, so every value is from the same instant:

```
// imu.h -- a register-level driver for an MPU-6050 style accelerometer + gyroscope, wri
// A Bus must provide:   bool write_reg(uint8_t dev, uint8_t reg, uint8_t value);
//                       bool read_regs(uint8_t dev, uint8_t reg, uint8_t* buf, uint8_t
#pragma once
#include <stdint.h>

namespace mpu {
const uint8_t ADDR = 0x68;           // the 7-bit I2C address with the AD0 pin low
const uint8_t REG_GYRO_CONFIG = 0x1B;
const uint8_t REG_ACCEL_CONFIG = 0x1C;
const uint8_t REG_ACCEL_XOUT_H = 0x3B;   // 14 bytes from here: accel X,Y,Z (H,L each),
const uint8_t REG_PWR_MGMT_1 = 0x6B;     // bit 6 = SLEEP, and it is SET after a power-u
const uint8_t REG_WHO_AM_I = 0x75;       // always reads 0x68
const uint8_t WHO_AM_I_VALUE = 0x68;
}  // namespace mpu

struct ImuSample { float ax_g, ay_g, az_g; float temp_c; float gx_dps, gy_dps, gz_dps; }

inline int16_t be16(uint8_t hi, uint8_t lo) { return (int16_t)(((uint16_t)hi << 8) | lo)

template <class Bus>
class Imu {
 public:
  enum Error : uint8_t { OK, NO_ANSWER, WRONG_DEVICE };
  explicit Imu(Bus& bus) : bus_(bus), accel_lsb_per_g_(16384.0f), gyro_lsb_per_dps_(131.

  // accel_range 0..3 = +-2, 4, 8, 16 g;  gyro_range 0..3 = +-250, 500, 1000, 2000 deg/s
  Error begin(uint8_t accel_range = 0, uint8_t gyro_range = 0) {
    uint8_t who = 0;
    if (!bus_.read_regs(mpu::ADDR, mpu::REG_WHO_AM_I, &who, 1)) return NO_ANSWER;
    if (who != mpu::WHO_AM_I_VALUE) return WRONG_DEVICE;
    if (!bus_.write_reg(mpu::ADDR, mpu::REG_PWR_MGMT_1, 0x00)) return NO_ANSWER;
    if (!bus_.write_reg(mpu::ADDR, mpu::REG_ACCEL_CONFIG, (uint8_t)((accel_range & 3) <<
    if (!bus_.write_reg(mpu::ADDR, mpu::REG_GYRO_CONFIG, (uint8_t)((gyro_range & 3) << 3
    static const float ACCEL_LSB[4] = {16384.0f, 8192.0f, 4096.0f, 2048.0f};
    static const float GYRO_LSB[4] = {131.0f, 65.5f, 32.8f, 16.4f};
    accel_lsb_per_g_ = ACCEL_LSB[accel_range & 3];
    gyro_lsb_per_dps_ = GYRO_LSB[gyro_range & 3];
    return OK;
  }

  bool read(ImuSample& s) {                                  // ONE burst of 14 bytes:
    uint8_t b[14];
    if (!bus_.read_regs(mpu::ADDR, mpu::REG_ACCEL_XOUT_H, b, 14)) return false;
```

```
    s.ax_g = be16(b[0], b[1]) / accel_lsb_per_g_;
    s.ay_g = be16(b[2], b[3]) / accel_lsb_per_g_;
    s.az_g = be16(b[4], b[5]) / accel_lsb_per_g_;
    s.temp_c = be16(b[6], b[7]) / 340.0f + 36.53f;
    s.gx_dps = be16(b[8], b[9]) / gyro_lsb_per_dps_;
    s.gy_dps = be16(b[10], b[11]) / gyro_lsb_per_dps_;
    s.gz_dps = be16(b[12], b[13]) / gyro_lsb_per_dps_;
    return true;
  }

 private:
  Bus& bus_;
  float accel_lsb_per_g_, gyro_lsb_per_dps_;
};
```

A fake MPU-6050 (a few dozen lines, only the registers the driver touches, with the sleep behaviour of the map) and four experiments:

```
// e_imu.cpp -- the register-level driver against a FAKE MPU-6050 that implements only t
// The fake is written from my reading of the register map; it shows that the driver doe
#include <cstdio>
#include <cmath>
#include <cstring>
#include "imu.h"

struct FakeMpu {
  uint8_t reg[128];
  uint8_t who;                                          // what WHO_AM_I will say (
  bool present;
  float ax, ay, az, temp, gx, gy, gz;                   // the "physical" truth: g,
  FakeMpu() : who(0x68), present(true), ax(-0.25f), ay(0.10f), az(0.95f), temp(25.0f), g
    std::memset(reg, 0, sizeof reg);
    reg[0x6B] = 0x40;                                   // power-up value: SLEEP bi
  }
  static void put(uint8_t* p, long v) { p[0] = (uint8_t)((v >> 8) & 0xFF); p[1] = (uint8
  uint8_t read1(uint8_t r) {
    if (r == 0x75) return who;
    if (r >= 0x3B && r <= 0x48) {
      if (reg[0x6B] & 0x40) return 0;                   // asleep: the data registe
      static const float ACC[4] = {16384.0f, 8192.0f, 4096.0f, 2048.0f}, GYR[4] = {131.0
      float a = ACC[(reg[0x1C] >> 3) & 3], g = GYR[(reg[0x1B] >> 3) & 3];
      uint8_t d[14];
      put(d + 0, std::lround(ax * a)); put(d + 2, std::lround(ay * a)); put(d + 4, std::
```

```
      put(d + 6, std::lround((temp - 36.53f) * 340.0f));
      put(d + 8, std::lround(gx * g)); put(d + 10, std::lround(gy * g)); put(d + 12, std
      return d[r - 0x3B];
    }
    return reg[r & 127];
  }
  bool write_reg(uint8_t dev, uint8_t r, uint8_t v) { if (!present || dev != 0x68) return
  bool read_regs(uint8_t dev, uint8_t r, uint8_t* buf, uint8_t n) {
    if (!present || dev != 0x68) return false;
    for (uint8_t i = 0; i < n; ++i) buf[i] = read1((uint8_t)(r + i));
    return true;
  }
};

static void show(const char* label, const ImuSample& s) {
  std::printf("  %-34s a = (%+.3f, %+.3f, %+.3f) g   T = %.2f C   w = (%+.2f, %+.2f, %+.
}

int main() {
  ImuSample s;
  {
    FakeMpu chip;
    std::printf("1. read the data registers WITHOUT waking the chip:\n");
    Imu<FakeMpu> imu(chip);
    imu.read(s);
    show("after power-up, still asleep", s);
  }
  {
    FakeMpu chip;
    Imu<FakeMpu> imu(chip);
    std::printf("2. begin(), then read (truth: a = (-0.250, 0.100, 0.950) g, T = 25 C, w
    int err = (int)imu.begin();                              // a separate statement: th
    std::printf("  begin() returned %d (0 = OK); PWR_MGMT_1 is now 0x%02X\n", err, chip.
    imu.read(s);
    show("+-2 g, +-250 deg/s", s);
  }
  {
    FakeMpu chip;
    Imu<FakeMpu> imu(chip);
    imu.begin(3, 3);                                         // +-16 g, +-2000 deg/s
    imu.read(s);
    show("+-16 g, +-2000 deg/s (range set)", s);
  }
  std::printf("3. errors the driver can report:\n");
  {
```

```
    FakeMpu chip;
    chip.present = false;
    Imu<FakeMpu> imu(chip);
    std::printf("  no chip on the bus:           begin() = %d (1 = NO_ANSWER)\n", (int)i
    FakeMpu other;
    other.who = 0x70;
    Imu<FakeMpu> imu2(other);
    std::printf("  a different chip (WHO_AM_I 0x70): begin() = %d (2 = WRONG_DEVICE)\n",
  }
  std::printf("4. two classic mistakes, with the same chip and the same truth:\n");
  {
    FakeMpu chip;
    Imu<FakeMpu> imu(chip);
    imu.begin();
    uint8_t b[14];
    chip.read_regs(0x68, 0x3B, b, 14);
    uint16_t wrong = ((uint16_t)b[0] << 8) | b[1];          // the same two bytes as an
    std::printf("  ax raw bytes 0x%02X 0x%02X: signed %d -> %+.3f g;  as unsigned %u -> %
                b[0], b[1], (int)be16(b[0], b[1]), be16(b[0], b[1]) / 16384.0f, (unsigne
    Imu<FakeMpu> forgot(chip);                              // the chip is set to +-16 g
    chip.reg[0x1C] = 3 << 3;
    forgot.read(s);
    show("chip at +-16 g, driver at +-2 g", s);
  }
  return 0;
}
```

```
1. read the data registers WITHOUT waking the chip:
  after power-up, still asleep       a = (+0.000, +0.000, +0.000) g   T = 36.53 C   w =
2. begin(), then read (truth: a = (-0.250, 0.100, 0.950) g, T = 25 C, w = (3.0, -7.5, -1
  begin() returned 0 (0 = OK); PWR_MGMT_1 is now 0x00
  +-2 g, +-250 deg/s                 a = (-0.250, +0.100, +0.950) g   T = 25.00 C   w =
  +-16 g, +-2000 deg/s (range set)   a = (-0.250, +0.100, +0.950) g   T = 25.00 C   w =
3. errors the driver can report:
  no chip on the bus:           begin() = 1 (1 = NO_ANSWER)
  a different chip (WHO_AM_I 0x70): begin() = 2 (2 = WRONG_DEVICE)
4. two classic mistakes, with the same chip and the same truth:
  ax raw bytes 0xF0 0x00: signed -4096 -> -0.250 g;  as unsigned 61440 -> +3.750 g (the
  chip at +-16 g, driver at +-2 g    a = (-0.031, +0.013, +0.119) g   T = 25.00 C   w =
```

Read the output as four lessons, each a mistake that a datasheet's register map predicts:

- **The chip starts asleep.** Experiment 1 skips the wake-up, and every value is exactly zero. Note the temperature: 36.53 °C, because a raw zero in the temperature formula is a believable temperature. A zero that reads as a plausible number is more dangerous than an obvious fault, so the driver's `begin()` clears the SLEEP bit before anything is read.
- **Verify the identity.** Experiment 3 shows `begin()` telling apart *no answer* from *another chip at the same address* (WHO_AM_I 0x70), which are different wiring faults.
- **Bytes are signed.** The same two bytes (0xF0, 0x00) are -4096, or -0.250 g, as a signed number and 61440, or +3.750 g, as an unsigned one: the sign is gone and the robot would believe it had a 3.75 g shove. `be16()` builds a signed 16-bit value on purpose.
- **The scale goes with the range.** Setting the chip to ±16 g and dividing by the ±2 g constant reads (-0.031, 0.013, 0.119) g where the truth is (-0.250, 0.100, 0.950): eight times too small. Experiment 2 also shows the price of range: at ±2000 deg/s the 3.0 deg/s turn reads 2.99, because a count is 0.061 deg/s there and 3.0 is not a whole number of counts.

This is a driver tested against *my reading* of the map. A real chip can also disagree with its own datasheet (a different silicon revision, a clone with another WHO_AM_I), and a real bus can return an error mid-transfer; the driver reports the second by returning `false`, and the sketch logs it, but nothing here tested it on hardware.

8. Wiring the build robot

The robot of this course has two driven wheels with an H-bridge motor driver for each (a speed pin and a direction pin), an ultrasonic sensor, two wheel encoders, a bumper switch, a battery-voltage divider and an IMU. The plan in section 2 gives each a pin, and the reasoning is in the table:

PART	PIN	WHY THAT PIN
left motor speed / direction	5 / 4	5 is PWM (Timer0 B); 4 is plain digital
right motor speed / direction	6 / 7	6 is PWM (Timer0 A); 7 is plain digital. Both motors share a timer, so they share a frequency
ultrasonic trigger / echo	8 / 12	any digital pins; 9 and 10 stay free for Timer1 and a servo later

wheel encoders	2 and 3	the only two interrupt-capable pins; reserved now, used in Programming for Robotics with Arduino C++ 4
bumper switch	11	digital input with the internal pull-up, so the switch just connects the pin to ground
battery voltage	A0 (14)	an analog input, through a divider so 5 V is never exceeded
IMU (SDA / SCL)	A4 / A5	the fixed I2C pins
USB serial	0 and 1	reserved: anything wired here disturbs uploads and `Serial`

The sketch below is the whole of it. `PinHW` has the same two functions as chapter 1's `SimHW`, so `WallRobot<PinHW>` is the *same robot code* with real pins. The echo state machine is polled on **every pass** of `loop()` because it needs fine time resolution, while everything else hangs off the scheduler. The bumper is read on every pass as well and **latches** an emergency stop, so the controller can no longer move the robot: an earlier draft only wrote a speed of zero, which the next 20 ms `act` would simply have overwritten.

```
// robot_ch2.ino -- Programming for Robotics with Arduino C++, chapter 2: the wall robot
// Compile-checked for the Arduino Uno R3 (arduino:avr:uno). NOT run on a board: no boar
#include <Wire.h>
#include "scheduler.h"
#include "log.h"
#include "robot.h"
#include "conv.h"
#include "echo.h"
#include "imu.h"
#include "pins.h"

const float V_MAX_MPS = 0.5f;                          // the speed that a PWM count of

// ---- the hardware as WallRobot sees it: the same two functions as SimHW in chapter 1
class PinHW {
 public:
  PinHW() : range_m_(9.99f), estop_(false) {}
  float range_m() { return range_m_; }
  void drive(float v) {                                // forward only in this chapter;
    uint8_t duty = estop_ ? 0 : speed_to_pwm(v, V_MAX_MPS);   // once the bumper has lat
    analogWrite(PIN_ML_PWM, duty);
```

```
    analogWrite(PIN_MR_PWM, duty);
  }
  void set_range(float m) { range_m_ = m; }
  void emergency_stop() { estop_ = true; drive(0.0f); }       // latched until the board
 private:
  float range_m_;
  bool estop_;
};

// ---- the I2C bus as Imu<> wants it ----
struct WireBus {
  bool write_reg(uint8_t dev, uint8_t reg, uint8_t value) {
    Wire.beginTransmission(dev);
    Wire.write(reg);
    Wire.write(value);
    return Wire.endTransmission() == 0;
  }
  bool read_regs(uint8_t dev, uint8_t reg, uint8_t* buf, uint8_t n) {
    Wire.beginTransmission(dev);
    Wire.write(reg);
    if (Wire.endTransmission(false) != 0) return false;           // false = keep the bu
    if (Wire.requestFrom(dev, n) != n) return false;
    for (uint8_t i = 0; i < n; ++i) buf[i] = Wire.read();
    return true;
  }
};

static PinHW hw;
static WallRobot<PinHW> robot(hw, 0.25f, 1.5f, V_MAX_MPS);
static Echo echo;
static WireBus wire_bus;
static Imu<WireBus> imu(wire_bus);
static ImuSample imu_sample;
static Scheduler<7> sch;
static RingLog<32> blackbox;
static bool trig_level = false;
static uint16_t battery_mv = 0;

enum { MSG_BOOT = 0, MSG_RANGE = 1, MSG_NO_ECHO = 2, MSG_IMU_FAIL = 3, MSG_BATT = 4 };

static void echo_start_task(void*, uint32_t) { echo.start(); }             // one measur
static void imu_task(void*, uint32_t now) { if (!imu.read(imu_sample)) blackbox.add(now,
static void battery_task(void*, uint32_t now) {
  battery_mv = counts_to_mv(analogRead(PIN_BATT));
  blackbox.add(now, LOG_DEBUG, MSG_BATT, (int16_t)battery_mv);
```

```
}
static void report_task(void*, uint32_t now) {
  Serial.print(now); Serial.print(F(" ms  range ")); Serial.print(robot.range(), 3);
  Serial.print(F(" m  cmd ")); Serial.print(robot.command(), 3);
  Serial.print(F(" m/s  az ")); Serial.print(imu_sample.az_g, 2);
  Serial.print(F(" g  batt ")); Serial.print(battery_mv); Serial.println(F(" mV"));
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT);
  digitalWrite(PIN_ML_DIR, HIGH); digitalWrite(PIN_MR_DIR, HIGH);             // forward
  pinMode(PIN_ML_PWM, OUTPUT); pinMode(PIN_MR_PWM, OUTPUT);
  pinMode(PIN_TRIG, OUTPUT); pinMode(PIN_ECHO, INPUT);
  pinMode(PIN_BUMP, INPUT_PULLUP); pinMode(PIN_LED, OUTPUT);
  Wire.begin();
  Wire.setClock(400000);
  Imu<WireBus>::Error e = imu.begin();
  blackbox.add(millis(), e == Imu<WireBus>::OK ? LOG_INFO : LOG_WARN, MSG_BOOT, (int16_t
  sch.add(WallRobot<PinHW>::sense_task, &robot, 20);
  sch.add(WallRobot<PinHW>::think_task, &robot, 20);
  sch.add(WallRobot<PinHW>::act_task, &robot, 20);
  sch.add(echo_start_task, 0, 60);
  sch.add(imu_task, 0, 20);
  sch.add(battery_task, 0, 1000);
  sch.add(report_task, 0, 500);
}

void loop() {
  // the echo state machine is polled on EVERY pass: it needs microsecond-level looks, t
  bool before = trig_level;
  Echo::Status st = echo.update(micros(), digitalRead(PIN_ECHO), trig_level);
  if (trig_level != before) digitalWrite(PIN_TRIG, trig_level);
  if (st == Echo::READY) hw.set_range(echo_us_to_m(echo.width_us()));
  else if (st == Echo::TIMEOUT) { hw.set_range(9.99f); blackbox.add(millis(), LOG_WARN, M
  if (digitalRead(PIN_BUMP) == LOW) hw.emergency_stop();                    // the bump
  sch.run(millis());
}
```

9. The budget

The compiler's report for `robot_ch2.ino` on the Uno, from the figures in this chapter:

BUILD	FLASH (OF 32,256)	RAM BY GLOBALS (OF 2,048)	LEFT FOR THE STACK
Programming for Robotics with Arduino C++ 1 sketch (log of 16)	4950 bytes (15 %)	473 bytes (23 %)	1575 bytes
`robot_ch2.ino`: Serial, Wire, 7 tasks, log of 32, echo and IMU drivers	9606 bytes (29 %)	936 bytes (45 %)	**1112 bytes**

Adding real hardware roughly doubled the flash (4950 to 9606) and the RAM (473 to 936). Of the RAM, `Serial` accounts for about 179 bytes, `Wire` for about 216, the 32-record log for 256 (8 bytes each, as in chapter 1), and the rest is the scheduler's seven tasks and the drivers. Flash still has 70 % free; RAM has 1112 bytes left for the stack, a comfortable margin only until a deeper call chain or a bigger buffer arrives, and, as in chapter 1, that figure is what is *available*: the stack's real depth was not measured because the sketch was not run.

The **time** budget for one 20 ms control period, by arithmetic from this chapter (the model, not a measurement of a board): one 14-byte IMU read at 400 kHz is 365 microseconds (1.8 %); one ADC reading is 104 microseconds (0.5 %); the echo state machine costs only its polls; and a 40-character status line at 115200 baud is 3.47 ms of wire time, which does not block while it fits in the 64-byte buffer. The scheduler and `Periodic` from chapter 1 measure the lateness of each task, and the first thing to do on a real board is to read those numbers.

Challenges

CHALLENGE 1: HOW FAST MUST THE ECHO BE POLLED?

Using the `Echo` state machine and a fake sensor, find how often it must be polled for the distance error to stay under 10 mm. Test intervals of 20, 40, 50, 58, 60, 80 and 200 microseconds, over 67 distances from 0.10 to 3.40 m, and vary where the polls fall in time relative to the sensor (8 phases). Compare the answer with the arithmetic (10 mm ÷ 0.1715 mm per microsecond).

View solution

CHALLENGE 2: HOW MUCH AVERAGING CAN THE LOOP AFFORD?

For ADC noise of 0.3, 1.5 and 4.0 counts (assumed), find the smallest number of readings (a power of two) whose average has a spread under 1 mV, and its cost in time at 104 microseconds a reading as a

share of a 20 ms period. Say what the third case means for the design.

View solution

CHALLENGE 3: WHEN DOES A LOG LINE STALL THE ROBOT?

Model the Uno's 64-byte transmit buffer (it accepts 64 bytes without waiting, then `print()` waits for space, and the wire drains one byte per 10 bit-times). A task prints a line of L bytes every 20 ms for two seconds. For 9600, 57600 and 115200 baud and L of 20, 40, 64, 100 and 200 bytes, report the longest single wait and the share of real time spent waiting.

View solution

QUICK REFERENCE

- **Pins.** A digital pin is a bit in `DDRx`, `PORTx`, `PINx`. A port toggle compiles to `sbi`/`cbi` (450 bytes of flash for the sketch); `digitalWrite` costs 734 bytes (+290), its specialised copy is 144 bytes and 66 instructions. No speed on a board was measured.
- **Pin table.** Six PWM pins (3, 5, 6, 9, 10, 11), two interrupt pins (2, 3), pins 0 and 1 are USB serial, A4 and A5 are I2C. A `constexpr` table with `static_assert` makes a wrong pin a build error.
- **PWM.** 976.56 Hz (Timer0, pins 5 and 6), 490.20 Hz (Timers 1 and 2). Prescaler 1 on Timer1 phase-correct gives 31,372.55 Hz. Timer0 also keeps `millis()`: do not change its prescaler. One duty step is 1.961 mm/s of 0.5 m/s; round (worst 0.980 mm/s), do not truncate (worst 1.941, always low).
- **ADC.** 4.888 mV a count; 104 us a reading (9615 a second). Averaging 64 readings cut an assumed 1.5-count noise to 0.93 mV in 6.66 ms; it needs noise to work. Integer conversion vs float: 1118 vs 1818 bytes of flash.
- **Echo.** `pulseIn` blocks for the whole echo (5.84 ms at 1 m, 20.42 ms at 3.5 m, its timeout with nothing there). The state machine polled every 100 us has a worst error of 17.0 mm (fake sensor).
- **Buses.** 115200 baud is +2.12 % in double-speed mode (the core's choice). A 40-character line is 3.47 ms at 115200 and 41.67 ms at 9600. 14 bytes over I2C: 1460 us at 100 kHz, 365 us at 400 kHz. Libraries: `Serial` +179 B RAM, `Wire` +216 B RAM (+2432 B flash), SPI +1 B RAM.
- **IMU.** The chip starts asleep (all zeros, temperature 36.53 °C); check WHO_AM_I; bytes are signed; the scale follows the range. Tested against a fake chip only.
- **Budget.** `robot_ch2.ino`: 9606 bytes flash (29 %), 936 bytes RAM (45 %), 1112 bytes available to the stack.
- **Challenges.** Echo under 10 mm needs polling every 58 us or faster (9.8 mm), 60 us gives 10.1; averaging to 1 mV needs 8, 64 or 512 readings for noise of 0.3, 1.5 and 4.0 counts (0.83, 6.66, 53.25

ms); at 9600 baud a 40-byte line every 20 ms makes `print()` wait 21.67 ms each time.

- **Not run:** anything on a board. Chip facts come from my memory of the datasheets; check them.

Next: Programming for Robotics with Arduino C++ 3, A Simulator to Test In

Programming for Robotics with Arduino C++ 3 (planned; not yet written) builds the thing that has been standing in for the world all along: a small C++ simulator of a two-wheel robot on the PC, with noisy range sensors and encoders, behind the same hardware interface as `PinHW`, so that the robot code in this chapter can be compiled for the board and for the host and run against the same recorded worlds.

Programming for Robotics with Arduino C++

Course 2 · Chapter 3 · A Simulator to Test In

Programming for Robotics with Arduino C++ 2 ended with a sketch that nobody has run: it compiles for the Uno, and that is all anyone knows. A robot cannot be debugged by wiring it and hoping. Every hardware mistake costs a minute of re-uploading, a noisy sensor never gives the same reading twice, and a real wall is not a place to find out that a controller overshoots. This chapter builds the thing that stands in for the world: a small **simulator** that runs on the PC, behind the *same* hardware interface as the real pins, so that the robot code is tested before it is flashed.

The chapter is about two things at once: how the simulator works, and, more important, **how far to trust it**. A simulator that is never tested against anything is a way of producing confident numbers. So the simulator is tested first against answers worked out by hand, then used to find three things about the robot that a first look at the code does not show, and then its limits are stated plainly.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`, no warnings): the simulator, its tests, every experiment with an output under it, and the challenge solutions. The simulator is a *model with assumed parameters* (motor lag, noise, dropouts, wheel size); none of them was measured on a real robot, so results show how the *code* behaves in a stated world, not how a real robot would behave.
> **Compiled for the Uno, not run** (arduino-cli 1.6.0, core `arduino:avr` 1.8.8): `robot_ch3.ino`. The flash and RAM figures are the compiler's; the sketch's encoders are not wired (they need interrupts, chapter 4), so on the board the odometry would stand still.
> **No board, motor, encoder or sensor was attached.** Nothing in this chapter was measured on hardware.

1. What a simulator is for, and what it cannot say

A simulator earns its place in four ways. It is **fast**: compiling and running the 2,400 twelve-second runs of section 6 took 1.4 seconds in total on this PC. It is **repeatable**: the same seed gives the same noise, so a failure can be reproduced exactly. It **knows the truth**: the robot's belief can be compared with where it really is, which no real robot can do. And it is **safe**: it can drive a robot into a wall a thousand times.

It cannot tell you what you did not put into it. A simulator contains the physics its author thought of, and a robot fails mostly for reasons the author did not think of. The right use is therefore modest: to find the bugs and the misunderstandings in *your code* cheaply, to make comparisons (this gain against that one), and to build a test that keeps a fixed problem fixed. It is not a replacement for putting the robot on the floor; it is what makes the first floor test a short one.

2. The seam: four functions

The robot code of this course does not call `analogWrite` or `digitalRead`. It calls four functions on an object the code is a template of, the `HW` parameter, exactly as `WallRobot<HW>` did in Programming for Robotics with Arduino C++ 1:

FUNCTION	MEANING
`float range_m()`	distance ahead in metres; **negative means no echo**
`void read_ticks(int32_t& l, int32_t& r)`	cumulative encoder ticks of each wheel, signed
`void drive(float vl, float vr)`	wheel surface speeds in m/s; negative means backwards
`bool bumped()`	the bumper switch

Anything with those four functions is a `HW`, and this chapter has five of them: `PinHW` (the real pins, section 8), `SimHW` (the simulator), `RecordingHW` (wraps another and writes down what the controller saw and did), `ReplayHW` (plays a recording back) and `AveragedRange` (wraps another and averages its range, in challenge 1). The controller does not know which one it has. Here it is in full: it is the part that will run on the board, and it is plain C++ with no Arduino call in it.

```
// drive.h -- the robot logic for a two-wheel robot that drives to a wall and stops shor
// It compiles unchanged for the Uno and for the PC. The hardware is a template parameter
//     float range_m();                              // distance ahead in metres; negati
//     void  read_ticks(int32_t& left, int32_t& right);   // cumulative encoder ticks (s
//     void  drive(float v_left, float v_right);     // wheel surface speeds, m/s (negat
//     bool  bumped();                               // true if the bumper is pressed
#pragma once
#include <stdint.h>
#include <math.h>                                  // sinf, cosf: on the Uno from avr

inline float clamp_f(float v, float lo, float hi) { return v < lo ? lo : (v > hi ? hi : v
```

```
enum DriveFault : uint8_t { FAULT_NONE = 0, FAULT_NO_RANGE = 1, FAULT_BUMPED = 2 };

struct DriveConfig {
  float stop_m;            // stop this far from the wall
  float k_range;           // speed per metre of distance still to go (1/s)
  float vmax;              // cruise speed (m/s)
  float k_heading;         // turn rate per radian of heading error (1/s); 0 switches he
  float wheel_r, track;    // metres
  int ticks_per_rev;
  uint8_t max_dropouts;    // this many consecutive "no echo" readings in a row is a fau
};

template <class HW>
class DriveRobot {
 public:
  DriveRobot(HW& hw, const DriveConfig& c)
      : hw_(hw), c_(c), range_(0.0f), have_range_(false), dropouts_(0), total_dropouts_(0
        x_(0.0f), y_(0.0f), th_(0.0f), tl_(0), tr_(0), first_ticks_(true), vl_(0.0f), vr_

  void sense(uint32_t) {
    float m = hw_.range_m();
    if (m < 0.0f) {                                          // no echo: keep
      ++dropouts_; ++total_dropouts_;
      if (dropouts_ > c_.max_dropouts) fault_ = FAULT_NO_RANGE;
    } else {
      dropouts_ = 0; range_ = m; have_range_ = true;
    }
    if (hw_.bumped()) fault_ = FAULT_BUMPED;
    int32_t l, r;
    hw_.read_ticks(l, r);
    if (first_ticks_) { tl_ = l; tr_ = r; first_ticks_ = false; }
    const float per_tick = 6.2831853f * c_.wheel_r / (float)c_.ticks_per_rev;
    float dl = (float)(l - tl_) * per_tick, dr = (float)(r - tr_) * per_tick;     // how
    tl_ = l; tr_ = r;
    float d = 0.5f * (dl + dr), dth = (dr - dl) / c_.track;
    // dead reckoning: advance along the middle of the turn
    float mid = th_ + 0.5f * dth;
#ifdef DRIVE_FAST_TRIG
    float m2 = mid * mid;                                    // first ter
    float c = 1.0f - 0.5f * m2, s = mid * (1.0f - m2 / 6.0f);
#else
    float c = cosf(mid), s = sinf(mid);                      // correct a
#endif
    x_ += d * c; y_ += d * s; th_ += dth;
```

```
  }

  void think(uint32_t) {
    if (fault_ != FAULT_NONE || !have_range_) { vl_ = vr_ = 0.0f; return; }   // no rang
    float v = clamp_f(c_.k_range * (range_ - c_.stop_m), 0.0f, c_.vmax);
    float w = -c_.k_heading * th_ * (v / c_.vmax);                            // hold the
    vl_ = v - 0.5f * w * c_.track;
    vr_ = v + 0.5f * w * c_.track;
  }

  void act(uint32_t) { hw_.drive(vl_, vr_); }

  float range() const { return range_; }
  float x() const { return x_; }
  float y() const { return y_; }
  float theta() const { return th_; }
  DriveFault fault() const { return fault_; }
  uint32_t dropouts() const { return total_dropouts_; }
  float command_left() const { return vl_; }
  float command_right() const { return vr_; }

  static void sense_task(void* c, uint32_t now) { static_cast<DriveRobot*>(c)->sense(now
  static void think_task(void* c, uint32_t now) { static_cast<DriveRobot*>(c)->think(now
  static void act_task(void* c, uint32_t now) { static_cast<DriveRobot*>(c)->act(now); }

 private:
  HW& hw_;
  DriveConfig c_;
  float range_;
  bool have_range_;
  uint8_t dropouts_;
  uint32_t total_dropouts_;
  DriveFault fault_;
  float x_, y_, th_;
  int32_t tl_, tr_;
  bool first_ticks_;
  float vl_, vr_;
};
```

Four design choices in it are worth noticing, each a lesson of an earlier chapter put to use. A reading of *no echo* keeps the last good range and is counted, and too many in a row is a **fault** that stops the robot, because an unreadable sensor must not be treated as a clear road. The speed is a clamped proportional of the distance still to go (it cannot go negative: the robot

never reverses). The heading hold fades out as the robot slows, so that it does not spin the robot on the spot at the wall. And the dead reckoning uses `sinf` and `cosf`, with a switch, `DRIVE_FAST_TRIG`, for a cheaper version that section 8 tests and shows to be unsafe at large angles.

3. The simulator

The simulator is a class with the same four functions plus a `step(dt)` that moves the world on. It is **host only**: it uses `<cmath>` and doubles and is never compiled for the Uno. What it models, and what it leaves out, is the list that decides how far to trust it:

MODELLED (ALL VALUES ARE ASSUMPTIONS)	NOT MODELLED
two wheels, 0.033 m radius, 0.15 m apart, a pose integrated every millisecond	wheel slip and stiction (a motor that will not turn below some duty)
motor speed that follows its command with a 0.1 s lag, an 8-bit duty (through chapter 2's `speed_to_pwm`), a weaker or stronger motor	battery voltage falling
encoders that count whole ticks (360 per wheel turn) from the true wheel travel	a sensor beam with width, or a wall at an angle that bounces the sound away
an ultrasonic range found by a ray to the nearest wall of a 3.0 × 2.0 m rectangle; Gaussian noise; readings with no echo; the echo width known only to the polling interval (chapter 2)	the I2C bus, the IMU, backlash
a seeded random generator, the same on every machine	variable loop timing: the control task runs exactly every 20 ms
	anything else in the room

```
// sim.h -- a small differential-drive simulator for the PC (HOST ONLY: it uses <cmath>,
// It stands behind the same hardware interface as the real pins: range_m(), read_ticks(
#pragma once
#include <stdint.h>
#include <cmath>
#include "conv.h"

#ifndef SIMREAL
#define SIMREAL double                                  // the simulator's own arithmetic
#endif
typedef SIMREAL real;

const double SIM_PI = 3.14159265358979323846;

// A tiny reproducible random generator: the same seed gives the same numbers on every m
```

```
struct Rng {
  uint32_t s;
  explicit Rng(uint32_t seed) : s(seed ? seed : 1u) {}
  uint32_t next() { s ^= s << 13; s ^= s >> 17; s ^= s << 5; return s; }          // xor
  double uniform() { return (next() >> 8) * (1.0 / 16777216.0); }                  // [
  double normal() {                                                                /
    double u1 = uniform();
    if (u1 < 1e-12) u1 = 1e-12;
    double u2 = uniform();
    return std::sqrt(-2.0 * std::log(u1)) * std::cos(2.0 * SIM_PI * u2);
  }
};

struct RobotParams {
  real wheel_r = 0.033;          // m
  real track = 0.15;             // m between the wheels
  int ticks_per_rev = 360;       // encoder ticks per wheel revolution
  real vmax = 0.5;               // wheel surface speed at a PWM duty of 255 (m/s), befo
  real tau = 0.10;               // motor lag, seconds: a wheel closes 63 % of a speed c
  real gain_l = 1.0, gain_r = 1.0;   // how strong each motor is: 0.95 means 5 % weaker
  bool pwm_quantise = true;      // true: commands go through speed_to_pwm (8-bit duty),
  real half_length = 0.06;       // from the centre to the sensor at the front
};

struct SensorParams {
  real sigma_m = 0.005;          // Gaussian noise on the distance (m): an ASSUMPTION, n
  real dropout = 0.02;           // probability that a reading returns no echo
  real poll_us = 100.0;          // the echo is measured by polling, so its width is a m
  real max_m = 4.0, min_m = 0.02;
};

struct Arena { real w = 3.0, h = 2.0; };               // a rectangle with a corner at

class SimHW {
 public:
  SimHW(const RobotParams& rp, const SensorParams& sp, const Arena& arena, uint32_t seed
      : x(0.3), y(1.0), th(0.0), rp_(rp), sp_(sp), arena_(arena), rng_(seed), wl_(0), wr_

  // ---- the hardware interface the robot code sees ----
  float range_m() {                                  // one new measurement per call;
    real d = ray_distance();
    if (d > sp_.max_m || d < sp_.min_m) return -1.0f;
    if (rng_.uniform() < sp_.dropout) return -1.0f;
    real noisy = d + sp_.sigma_m * rng_.normal();
    if (noisy < 0) noisy = 0;
```

```
    double us = (double)m_to_echo_us((float)noisy);
    us = std::floor(us / sp_.poll_us) * sp_.poll_us;   // the echo width is only known t
    return echo_us_to_m((uint32_t)us);
  }
  void read_ticks(int32_t& l, int32_t& r) const { l = tl_; r = tr_; }
  void drive(float vl, float vr) { cmd_l_ = vl; cmd_r_ = vr; }
  bool bumped() const { return ray_distance() <= 0.0; }

  // ---- physics: advance by dt seconds ----
  void step(real dt) {
    real target_l = target_speed(cmd_l_, rp_.gain_l), target_r = target_speed(cmd_r_, rp_
    if (rp_.tau > 0) { wl_ += (target_l - wl_) * dt / rp_.tau; wr_ += (target_r - wr_) *
    else { wl_ = target_l; wr_ = target_r; }
    sl_ += wl_ * dt;
    sr_ += wr_ * dt;
    real v = (wl_ + wr_) / 2, w = (wr_ - wl_) / rp_.track;
    real mid = th + w * dt / 2;                          // midpoint integration of the he
    x += v * std::cos(mid) * dt;
    y += v * std::sin(mid) * dt;
    th += w * dt;
    real per_tick = 2.0 * SIM_PI * rp_.wheel_r / rp_.ticks_per_rev;
    tl_ = (int32_t)std::floor(sl_ / per_tick);
    tr_ = (int32_t)std::floor(sr_ / per_tick);
    t += dt;
  }

  real ray_distance() const {                            // from the sensor at the front o
    real px = x + rp_.half_length * std::cos(th), py = y + rp_.half_length * std::sin(th
    if (px <= 0 || px >= arena_.w || py <= 0 || py >= arena_.h) return 0.0;
    real dx = std::cos(th), dy = std::sin(th), best = 1e9;
    if (dx > 1e-12) best = std::fmin(best, (arena_.w - px) / dx); else if (dx < -1e-12) b
    if (dy > 1e-12) best = std::fmin(best, (arena_.h - py) / dy); else if (dy < -1e-12) b
    return best;
  }

  const RobotParams& params() const { return rp_; }
  real x, y, th;                                          // the TRUTH: only the simulator
  real wheel_l() const { return wl_; }
  real wheel_r() const { return wr_; }
  real time() const { return t; }

 private:
  real target_speed(float cmd, real gain) const {
    real mag = cmd < 0 ? -cmd : cmd;
    real sign = cmd < 0 ? -1.0 : 1.0;
```

```
    if (rp_.pwm_quantise) return sign * speed_to_pwm((float)mag, (float)rp_.vmax) / 255.
    return cmd * gain;
  }
  RobotParams rp_;
  SensorParams sp_;
  Arena arena_;
  Rng rng_;
  real wl_, wr_;
  int32_t tl_, tr_;
  real sl_, sr_, t;
  float cmd_l_ = 0.0f, cmd_r_ = 0.0f;
};
```

One detail is easy to miss and mattered: the random generator is the simulator's own, a few lines of xorshift with a Box-Muller step for the Gaussian, not the C++ library's. The library's random distributions are allowed to give *different numbers on different compilers*, so a seed would not name one run everywhere. With the simulator's own generator a seed does, and challenge 3 uses that.

4. Test the simulator first

Before the simulator is allowed to say anything about the robot, it is checked against numbers worked out by hand: a straight run with motor lag, a quarter-circle arc, the encoder count for one metre of travel, the duty quantisation, the range to the walls in four directions, and the noise model.

```
physics:
  PASS straight 0.2 m/s for 2 s with 0.1 s motor lag: x     got    0.68020   want    0.6
  PASS ... and y stays                                      got    1.00000   want    1.0
  PASS quarter arc: heading (rad)                           got    1.57067   want    1.5
  PASS quarter arc: x = 0.3 + R sin(theta)                  got    0.52500   want    0.5
  PASS quarter arc: y = 1.0 + R (1 - cos(theta))            got    1.22497   want    1.2
  PASS encoder ticks after 1 m of wheel travel (left)       got 1736.00000   want 1736.0
  PASS encoder ticks after 1 m of wheel travel (right)      got 1736.00000   want 1736.0
  PASS 0.25 m/s through the 8-bit duty: wheel speed         got    0.25098   want    0.2
range sensor (geometry worked out by hand; the sensor sits 0.06 m in front of the centre
  PASS facing +x from (0.3, 1.0)                            got    2.64007   want    2.6
  PASS facing +y from (0.3, 1.0)                            got    0.93999   want    0.9
  PASS 45 degrees from (1.0, 0.5)                           got    2.06126   want    2.0
  PASS facing -x from (1.0, 0.5)                            got    0.93999   want    0.9
```

```
  noise model:
    PASS mean of 200000 readings of 2.64 m                  got    2.63999   want    2.64
    PASS spread of the readings (sigma 0.010 m)             got    0.00998   want    0.01
    PASS fraction of readings with no echo (10 %)           got    0.09823   want    0.10
    PASS poll 100 us: 2.64 m reads (a multiple of 17.15 mm) got    2.62395   want    2.62
    PASS ... and is below the truth by 0 to 17.15 mm (one step) got    0.01605   want    0
    PASS same seed: readings that agree out of 1000         got 1000.00000   want 1000.00
    PASS different seed: readings that agree out of 1000    got  211.00000   want  200.00
  0 failure(s)
```

All 19 checks pass, and each was set up so that a wrong simulator would fail. The straight run reaches x = 0.68020 where the lag formula ($x = x0 + v(t - \tau(1 - e^{-t/\tau}))$) gives 0.68000: the 0.2 mm is the millisecond steps. The arc ends at (0.52500, 1.22497) for a radius of 0.225 m, which is what the geometry says, and one metre of wheel travel is 1736 ticks, which is $1 / (2\pi \times 0.033) \times 360 = 1736.3$ by hand. The ranges are hand-geometry (the 45° case, 2.06117 m, I worked out on paper as (2 - 0.54243) / sin 45°). A tolerance of a millimetre or two is allowed where the hand arithmetic itself was rounded.

Two of the checks taught something. The **dropout fraction** came out at 9.5 % where 10 % was set, 2.3 standard errors low, and I did not simply widen the tolerance: I tested the random generator alone (12 seeds, 200,000 draws each, average 0.0999) and the sensor path alone (12 seeds, average 9.95 %), and the low result was seed 7 being an ordinary outlier. The test now uses ten times as many readings (9.82 %, inside a tolerance of 0.3 %). The second: with seeds 42 and 43 the readings *agree on 211 of 1000*, which looks like a bug until you remember that the range is quantised to 17 mm steps and a noise of 20 mm is only about one step wide, so different noise often gives the same reading. A test that expects different seeds to give different results must allow for what the quantiser throws away.

5. The robot in the simulator

The first experiment: from (0.30, 1.00), heading toward the wall at x = 3.0 m, with the stop line at 0.25 m. Five runs: ideal motors; then the right motor made 5 % weak, with the heading hold off and then at gains of 2, 6 and 20.

```
// e_drive.cpp -- the robot in the simulator: ideal motors, then a weak right motor with
#include <cstdio>
#include <initializer_list>
```

```
#include "harness.h"

static void show(const char* label, const Outcome& o, double start_y) {
  const char* f = o.fault == FAULT_NONE ? "none" : o.fault == FAULT_NO_RANGE ? "NO_RANGE
  std::printf("  %-30s  x %5.3f  y %6.3f  drift %+7.1f mm  heading %+7.2f deg  odometry %
              label, o.x, o.y, (o.y - start_y) * 1000.0, o.th * 180.0 / SIM_PI, o.odo_th
}

int main() {
  RobotParams rp;
  SensorParams sp;                                    // 5 mm noise, 2 % dropouts, poll
  Arena arena;
  DriveConfig cfg = default_config();
  std::printf("from (0.30, 1.00) heading +x toward the wall at x = 3.0 m, stop line 0.25
  {
    SimHW w(rp, sp, arena, 1);
    show("ideal motors, hold 2.0", run_robot(w, cfg, 10000), 1.0);
  }
  RobotParams weak = rp;
  weak.gain_r = 0.95;                                 // the right motor is 5 % weaker
  for (float k : {0.0f, 2.0f, 6.0f, 20.0f}) {
    DriveConfig c = cfg; c.k_heading = k;
    SimHW w(weak, sp, arena, 1);
    char label[64];
    std::snprintf(label, sizeof label, "weak right motor, hold %.1f", k);
    show(label, run_robot(w, c, 10000), 1.0);
  }
  return 0;
}
```

```
from (0.30, 1.00) heading +x toward the wall at x = 3.0 m, stop line 0.25 m, 10 s, seed 1
  ideal motors, hold 2.0          x 2.690  y  1.000  drift    +0.0 mm  heading   +0.00 d
  weak right motor, hold 0.0      x 2.298  y  0.212  drift  -788.5 mm  heading  -43.08 d
  weak right motor, hold 6.0      x 2.690  y  0.878  drift  -122.0 mm  heading   -0.69 d
  weak right motor, hold 20.0     x 2.690  y  0.964  drift   -36.5 mm  heading   -0.06 d
```

Three things in the output are not obvious from the code.

- **The robot can stop in the right place at the wrong wall.** With a weak motor and no heading hold the robot veered to -43°, drifted 788.5 mm sideways and stopped *exactly* 0.250 m from a wall, true range 0.250, no fault. It was 0.25 m from the bottom wall, because the

range sensor only measures along the heading and cannot tell which wall it faces. Both the range controller and the sensor were working perfectly. The thing that failed is that nothing was holding the heading.

- **The odometry knew.** Its heading, -43.12°, matches the truth, -43.08°: the encoders saw the drift that the range sensor could not. That is the reason to have encoders.
- **A proportional hold reduces the drift but cannot remove it.** A correction proportional to the heading error needs an error to exist before it produces any correction, so a steady disturbance (a weak motor) leaves a steady error. The drift fell from 788.5 mm (no hold) to 318.8, 122.0 and 36.5 mm at gains of 2, 6 and 20, roughly in inverse proportion to the gain (drift times gain is 638, 732 and 730). A bigger gain helps, with a limit set by the loop's period and the motor's lag that none of these runs reached: nothing misbehaved even at 20, which is a fact about this model, not a promise. The cure for a steady error, an integral term, comes with PID in a later chapter.

6. Noise: 200 seeded runs per setting

A single run says little about a noisy system. Each row below is 200 runs with seeds 1 to 200, with the error measured as the true distance to the wall at the end minus the 250 mm stop line (positive means stopped short, negative means went past). Runs that ended in a no-range fault are counted but not averaged, because a robot standing still far from the wall has not "stopped at the line".

```
200 runs each (seeds 1..200), ideal motors, heading hold on; error = true distance to th
  setting                          runs   fault    bump    mean mm    sd mm     min mm
  noise and dropouts, echo polled every 100 us:
  sigma  0 mm, dropouts  0 %          200       0       0       +5.8      0.0       +5.8
  sigma  0 mm, dropouts  5 %          200       0       0       +5.8      0.1       +5.6
  sigma  0 mm, dropouts 30 %          200      44       0       +5.7      0.2       +5.1
  sigma  5 mm, dropouts  0 %          200       0       0       -1.4      0.5       -2.7
  sigma  5 mm, dropouts  5 %          200       0       0       -1.4      0.6       -3.1
  sigma  5 mm, dropouts 30 %          200      44       0       -1.5      0.7       -3.6
  sigma 20 mm, dropouts  0 %          200       0       0      -17.5      1.6      -22.2
  sigma 20 mm, dropouts  5 %          200       0       0      -17.4      1.8      -22.8
  sigma 20 mm, dropouts 30 %          200      44       0      -17.6      1.9      -22.9
  echo poll interval, sigma 5 mm, dropouts 2 %:
  polled every    1 us (step   0.2 mm)  200       0       0       -5.6      0.5       -6
  polled every   10 us (step   1.7 mm)  200       0       0       -4.9      0.5       -6
```

```
  polled every  100 us (step  17.2 mm)  200       0       0       -1.4      0.5       -2
  polled every 1000 us (step 171.5 mm)  200       0       0      +75.9      2.0      +70
```

Read the table in three parts.

- **Dropouts do not move the answer, they stop the job.** At 30 % no-echo readings 44 of 200 runs (22 %) hit the fault: five consecutive readings with nothing is likely over 500 samples when each has a 30 % chance. The fault *works*, since the robot stands still and nothing hits the wall (no bumps in any row), but a standing robot is also a failed task. The count is the same in the three rows of 30 % because the seed fixes the dropout pattern whatever the noise level; that is exactly what a seed is for, and it makes the rows comparable.
- **Noise pushes the robot past the line, systematically.** The mean error goes from +5.8 mm with no noise to -1.4 mm at 5 mm of noise and -17.5 mm at 20 mm. The reason is the controller's one-way speed: once a noisy reading is large the robot moves forward, and it can never move back, so it ratchets toward the most extreme reading. That is a hypothesis until it is tested, so it was: the same experiment with the lower speed clamp changed from 0 to the negative of the top speed (the robot may reverse) gave **+7.3, +8.1 and +8.3 mm** at the same three noise levels, and +0.0 mm at the finest polling. The bias disappeared when reversing was allowed. It was the controller, not the sensor.
- **The polling interval sets a floor under the accuracy.** With echo polling every 1000 microseconds, one step is 171.5 mm: the sensor reports 0.1715 m or 0.343 m, never between, and the robot stops 75.9 mm short, each time, with a spread of only 2 mm. This is the polling error of Programming for Robotics with Arduino C++ 2 seen from the robot's side. The +5.8 mm of the no-noise row is the same effect at 17 mm steps: a reading is rounded *down*, which means measured distances are always a little short, so the robot thinks it is nearer than it is and stops early.

The first and third of these effects were predicted by the arithmetic of chapter 2; the second was not, and it is the sort of finding a simulator is for.

```
  setting                          runs   fault    bump    mean mm    sd mm     min mm
  noise and dropouts, echo polled every 100 us:
  sigma  0 mm, dropouts  0 %          200       0       0       +7.3      0.0       +7.3
  sigma  5 mm, dropouts  0 %          200       0       0       +8.1      0.9       +5.2
  sigma 20 mm, dropouts  0 %          200       0       0       +8.3      2.5       +0.6
  echo poll interval, sigma 5 mm, dropouts 2 %:
  polled every    1 us (step   0.2 mm)  200       0       0       +0.0      0.6       -1
  polled every   10 us (step   1.7 mm)  200       0       0       +0.8      0.6       -0
```

```
  polled every  100 us (step  17.2 mm)  200       0       0       +8.2      1.0       +5
  polled every 1000 us (step 171.5 mm)  200       0       0      +92.5      3.0      +83
```

(That table is the same program as the previous one, rebuilt with the lower speed clamp of `drive.h` changed from 0 to the negative of the top speed; only the rows that are comparable are shown.)

7. Replay: running a recorded run again

Some bugs only show up in a particular run. A **recording** keeps what the controller saw at every control tick (the range, the encoder counts, the bumper) and what it did (the two wheel commands and its odometry). The values are stored as hexadecimal floats, so nothing is rounded: reading the file back gives the same numbers bit for bit. **Replay** then feeds the recorded inputs to the controller again.

```
// replay.h -- record the sensor inputs and the outputs of a run to a text file, and fee
// A line per 20 ms control tick:  range  ticks_left  ticks_right  bumped  |  v_left  v_
#pragma once
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <vector>
#include "drive.h"

struct Tick {
  float range; int32_t tl, tr; int bump;              // inputs the controller saw
  float vl, vr, ox, oy;                               // outputs it produced
};

// Wraps any HW: forwards every call and notes what the controller saw and did.
template <class HW>
class RecordingHW {
 public:
  explicit RecordingHW(HW& hw) : hw_(hw) { cur_.range = 0; cur_.tl = cur_.tr = 0; cur_.bu
  float range_m() { cur_.range = hw_.range_m(); return cur_.range; }
  bool bumped() { cur_.bump = hw_.bumped(); return cur_.bump != 0; }
  void read_ticks(int32_t& l, int32_t& r) { hw_.read_ticks(l, r); cur_.tl = l; cur_.tr =
  void drive(float vl, float vr) { cur_.vl = vl; cur_.vr = vr; hw_.drive(vl, vr); }
  Tick cur_;                                          // the caller copies this after
 private:
```

```
  HW& hw_;
};

// Plays recorded inputs back, one tick at a time. drive() just notes the command.
class ReplayHW {
 public:
  explicit ReplayHW(const std::vector<Tick>& ticks) : vl_(0), vr_(0), t_(ticks), i_(0) {
  void set_index(size_t i) { i_ = i; }
  float range_m() { return t_[i_].range; }
  bool bumped() { return t_[i_].bump != 0; }
  void read_ticks(int32_t& l, int32_t& r) { l = t_[i_].tl; r = t_[i_].tr; }
  void drive(float vl, float vr) { vl_ = vl; vr_ = vr; }
  float vl_, vr_;
 private:
  const std::vector<Tick>& t_;
  size_t i_;
};

inline bool save_ticks(const char* path, const std::vector<Tick>& v) {
  FILE* f = std::fopen(path, "w");
  if (!f) return false;
  for (const Tick& t : v)
    std::fprintf(f, "%a %d %d %d | %a %a %a %a\n", (double)t.range, (int)t.tl, (int)t.tr
  std::fclose(f);
  return true;
}

inline bool load_ticks(const char* path, std::vector<Tick>& v) {
  FILE* f = std::fopen(path, "r");
  if (!f) return false;
  double r, vl, vr, ox, oy; int tl, tr, bump;
  while (std::fscanf(f, "%la %d %d %d | %la %la %la %la", &r, &tl, &tr, &bump, &vl, &vr,
    Tick t; t.range = (float)r; t.tl = tl; t.tr = tr; t.bump = bump; t.vl = (float)vl; t
    v.push_back(t);
  }
  std::fclose(f);
  return true;
}

inline uint32_t fnv1a(const void* data, size_t n, uint32_t h = 2166136261u) {   // a 32-
  const unsigned char* p = (const unsigned char*)data;
  for (size_t i = 0; i < n; ++i) { h ^= p[i]; h *= 16777619u; }
  return h;
}
```

Two runs were recorded, with the right motor weak: one with the heading hold at 2.0, the other with it off. Then they were replayed with the same code, with a changed range gain, and with a different build of the same controller, the cheap trigonometry:

```
--- record two runs (a weak right motor; heading gain 2.0 and 0.0)
recorded 500 ticks to run_hold.txt; fingerprint of the commands ce1530d6; true range at
recorded 500 ticks to run_free.txt; fingerprint of the commands 05fe9965; true range at
--- replay the same code on the same inputs
replayed 500 ticks of run_hold.txt (heading gain 2.0, k_range 1.50): fingerprint ce1530d
replayed 500 ticks of run_free.txt (heading gain 0.0, k_range 1.50): fingerprint 05fe996
--- replay with a changed gain (k_range 1.5 -> 1.2)
replayed 500 ticks of run_hold.txt (heading gain 2.0, k_range 1.20): fingerprint 865c632
--- replay with the cheap sin/cos (a different build of the same controller)
replayed 500 ticks of run_hold.txt (heading gain 2.0, k_range 1.50): fingerprint ce1530d
replayed 500 ticks of run_free.txt (heading gain 0.0, k_range 1.50): fingerprint 05fe996
--- the same changed gain run for real, in the closed loop (the world answers back)
closed loop, 500 ticks to -; fingerprint of the commands ce1530d6; true range at the end
closed loop, 500 ticks to -; fingerprint of the commands 04fbafcb; true range at the end
```

Four results, each one a use of replay or a limit on it.

- **Same code, same inputs, identical outputs.** The fingerprint of the 500 recorded commands (a 32-bit hash of their bytes) is `ce1530d6` when recorded and `ce1530d6` on replay, with 0 commands differing. This is the basis of a regression test: if a refactoring changes the fingerprint, it changed behaviour.
- **A different build can change the answer in a way the commands hide.** With the cheap sin and cos, replaying the free-running run gave 0 different commands, because the commands depend on the heading, which is not calculated with trigonometry, yet the odometry *position* differs by up to 5.8 mm. A test that compares only the commands would have passed a change that made the position estimate worse.
- **A changed gain is visible at once.** With the range gain at 1.2 instead of 1.5, 178 of 500 commands differed, from tick 216 (4.32 s, about when the robot starts to slow for the wall and the gain starts to matter), by up to 0.1048 m/s.
- **But replay cannot say what the changed controller would have *done*.** The recorded world did not answer the changed commands: the robot in the recording did not slow down earlier, so the next recorded range is the one the old controller produced. Running the changed gain for real, in the closed loop, gave a *different* fingerprint (`04fbafcb`) from the replay of the same change (`865c632a`), and a different ending (0.253 m from the wall, drift -313.9 mm,

against 0.251 m and -318.8 mm for the original). Replay tests the controller's decisions on fixed inputs; it does not test the loop, which needs the simulator.

8. One robot, two targets

The point of the seam is that the file that ran in the simulator is the file that goes on the board. That was checked as a fact, not assumed: the copy of `drive.h` in the sketch folder and the one used by the simulator have the same MD5 hash, as have `timing.h` and `scheduler.h`:

```
51f2295bc77c4be9fc7ad4aabe632c1f *drive.h
51f2295bc77c4be9fc7ad4aabe632c1f *robot_ch3/drive.h
b980062e47e755467479e3cbd042b012 *timing.h
b980062e47e755467479e3cbd042b012 *robot_ch3/timing.h
e385bc57a49aae463056a5e7e613eb19 *scheduler.h
e385bc57a49aae463056a5e7e613eb19 *robot_ch3/scheduler.h
```

The sketch is a thin shell: `PinHW` implements the four functions on pins (the range from chapter 2's echo state machine, the bumper from a pin, `drive` as direction lines and PWM), and `setup()` and `loop()` hand control to the scheduler. The encoders are not wired in this chapter, so `read_ticks` returns zero and says so.

```
// robot_ch3.ino -- Programming for Robotics with Arduino C++, chapter 3: the SAME drive
// Compile-checked for the Arduino Uno R3 (arduino:avr:uno). NOT run on a board: no boar
// The wheel encoders are not wired yet (they need interrupts: chapter 4), so read_ticks
#include "scheduler.h"
#include "conv.h"
#include "echo.h"
#include "pins.h"
#include "drive.h"

const float V_MAX_MPS = 0.5f;                          // wheel speed at a duty of 255: 

class PinHW {                                          // the four functions drive.h nee
 public:
  PinHW() : range_m_(-1.0f) {}
  float range_m() { return range_m_; }                 // negative until the first echo,
  void set_range(float m) { range_m_ = m; }
  void read_ticks(int32_t& l, int32_t& r) { l = 0; r = 0; }
  bool bumped() { return digitalRead(PIN_BUMP) == LOW; }
  void drive(float vl, float vr) {
```

```
    digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
    digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
    analogWrite(PIN_ML_PWM, speed_to_pwm(vl >= 0.0f ? vl : -vl, V_MAX_MPS));
    analogWrite(PIN_MR_PWM, speed_to_pwm(vr >= 0.0f ? vr : -vr, V_MAX_MPS));
  }
 private:
  float range_m_;
};

static PinHW hw;
static DriveConfig make_config() {
  DriveConfig c;
  c.stop_m = 0.25f; c.k_range = 1.5f; c.vmax = V_MAX_MPS; c.k_heading = 2.0f;
  c.wheel_r = 0.033f; c.track = 0.15f; c.ticks_per_rev = 360; c.max_dropouts = 5;
  return c;
}
static DriveRobot<PinHW> robot(hw, make_config());
static Echo echo;
static Scheduler<5> sch;
static bool trig_level = false;

static void echo_start_task(void*, uint32_t) { echo.start(); }
static void report_task(void*, uint32_t now) {
  Serial.print(now); Serial.print(F(" ms  range ")); Serial.print(robot.range(), 3);
  Serial.print(F(" m  cmd ")); Serial.print(robot.command_left(), 3);
  Serial.print(F(" m/s  fault ")); Serial.println((int)robot.fault());
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT);
  pinMode(PIN_ML_PWM, OUTPUT); pinMode(PIN_MR_PWM, OUTPUT);
  pinMode(PIN_TRIG, OUTPUT); pinMode(PIN_ECHO, INPUT);
  pinMode(PIN_BUMP, INPUT_PULLUP);
  sch.add(DriveRobot<PinHW>::sense_task, &robot, 20);
  sch.add(DriveRobot<PinHW>::think_task, &robot, 20);
  sch.add(DriveRobot<PinHW>::act_task, &robot, 20);
  sch.add(echo_start_task, 0, 60);
  sch.add(report_task, 0, 500);
}

void loop() {
  bool before = trig_level;
  Echo::Status st = echo.update(micros(), digitalRead(PIN_ECHO), trig_level);
  if (trig_level != before) digitalWrite(PIN_TRIG, trig_level);
```

```
  if (st == Echo::READY) hw.set_range(echo_us_to_m(echo.width_us()));
  else if (st == Echo::TIMEOUT) hw.set_range(-1.0f);
  sch.run(millis());
}
```

Compiled for the Uno, without warnings from the project's own files:

BUILD	FLASH (OF 32,256)	RAM BY GLOBALS (OF 2,048)
`robot_ch3.ino`, `sinf` and `cosf`	7726 bytes (23 %)	413 bytes (20 %)
`robot_ch3.ino`, `-DDRIVE_FAST_TRIG`	7454 bytes (23 %), 272 less	413 bytes (20 %)

The two figures are not comparable to chapter 2's sketch (9606 bytes, 936 bytes of RAM), because this sketch has no `Wire`, IMU or log. The cheap trigonometry saves 272 bytes of flash, and the next experiment is the price, the error of its two series against the library functions:

```
// e_trig.cpp -- the cheap sin/cos of the odometry (first terms of the series) against s
#include <cstdio>
#include <math.h>
#include <initializer_list>

int main() {
  std::printf("  heading rad   deg     sin approx   sin exact    error     cos approx   
  for (float a : {0.05f, 0.1f, 0.3f, 0.5f, 0.75f, 1.0f, 1.5f, 2.0f, 3.1416f}) {
    float m2 = a * a, c = 1.0f - 0.5f * m2, s = a * (1.0f - m2 / 6.0f);
    std::printf("  %11.4f  %5.1f   %+9.5f   %+9.5f   %+8.5f   %+9.5f   %+9.5f   %+8.5f\n
  }
  return 0;
}
```

```
  heading rad   deg     sin approx   sin exact    error     cos approx   cos exact    er
       0.0500    2.9    +0.04998    +0.04998   -0.00000    +0.99875    +0.99875   -0.000
       0.1000    5.7    +0.09983    +0.09983   -0.00000    +0.99500    +0.99500   -0.000
       0.3000   17.2    +0.29550    +0.29552   -0.00002    +0.95500    +0.95534   -0.000
       0.5000   28.6    +0.47917    +0.47943   -0.00026    +0.87500    +0.87758   -0.002
       0.7500   43.0    +0.67969    +0.68164   -0.00195    +0.71875    +0.73169   -0.012
       1.0000   57.3    +0.83333    +0.84147   -0.00814    +0.50000    +0.54030   -0.040
       1.5000   85.9    +0.93750    +0.99749   -0.05999    -0.12500    +0.07074   -0.195
```

```
        2.0000  114.6    +0.66667    +0.90930   -0.24263    -1.00000    -0.41615   -0.583
        3.1416  180.0    -2.02615    -0.00001   -2.02614    -3.93482    -1.00000   -2.934
```

At the headings of the weak-motor runs it is harmless: the cosine is wrong by 0.00034 at 17°, 0.0129 at 43°. By 86° the cosine has the wrong sign (-0.125 against +0.071), and at 180° it returns -3.93 for a function that cannot leave -1 to +1. A robot that turns round, which any robot with a planner will, would have its position estimate wrecked, silently. The default of this course is therefore the library; the cheap one is for a robot that can prove its heading stays small, and 272 bytes is 0.8 % of the flash. That is the budget lesson: **a saving is only a saving where its error is acceptable**, and the way to find out is to measure the error, not the flash.

9. Is the simulation reproducible?

If a seed is to be a bug report, the same seed must give the same run every time. Four things could break that: the compiler's optimisation, floating-point shortcuts, the number type used for the world, and the random generator. The experiment builds the same program at four optimisation levels, with the unsafe-looking `-ffast-math`, and with a `float` world, and compares the output byte for byte:

```
#!/bin/bash
# is the simulation reproducible? build e_drive at several optimisation levels and with a
GPP="C:/Users/emuba/AppData/Local/Microsoft/WinGet/Packages/BrechtSanders.WinLibs.POSIX.U
export PATH="$(cygpath -u "$(dirname "$GPP")"):$PATH"
cd "$(dirname "$0")"
echo "build                                  fingerprint of the output (md5, first 8 hex
for opt in -O0 -O1 -O2 -O3; do
  g++ -std=c++11 $opt e_drive.cpp -o det.exe && printf "%-38s  %s\n" "double world, $opt
done
g++ -std=c++11 -O2 -ffast-math e_drive.cpp -o det.exe && printf "%-38s  %s\n" "double wor
g++ -std=c++11 -O2 -DSIMREAL=float e_drive.cpp -o detf.exe && printf "%-38s  %s\n" "FLOAT
echo "--- the two outputs, line by line, where they differ (double world first, float wor
g++ -std=c++11 -O2 e_drive.cpp -o det.exe && diff <(./det.exe) <(./detf.exe) | cut -c1-20
```

```
build                                  fingerprint of the output (md5, first 8 hex)
double world, -O0                      0cfd4dfe
double world, -O1                      0cfd4dfe
double world, -O2                      0cfd4dfe
```

```
double world, -O3                       0cfd4dfe
double world, -O2 -ffast-math           0cfd4dfe
FLOAT world, -O2                        357e0779
--- the two outputs, line by line, where they differ (double world first, float world se
3,6c3,6
<   weak right motor, hold 0.0      x 2.298  y  0.212  drift  -788.5 mm  heading  -43.08
<   weak right motor, hold 2.0      x 2.691  y  0.681  drift  -318.8 mm  heading   -5.56
<   weak right motor, hold 6.0      x 2.690  y  0.878  drift  -122.0 mm  heading   -0.69
<   weak right motor, hold 20.0     x 2.690  y  0.964  drift   -36.5 mm  heading   -0.06
---
>   weak right motor, hold 0.0      x 2.298  y  0.212  drift  -788.5 mm  heading  -43.08
>   weak right motor, hold 2.0      x 2.691  y  0.681  drift  -319.1 mm  heading   -5.59
>   weak right motor, hold 6.0      x 2.690  y  0.878  drift  -122.3 mm  heading   -0.74
>   weak right motor, hold 20.0     x 2.690  y  0.963  drift   -36.9 mm  heading   -0.09
```

On this machine and this compiler the output is **bit-identical from `-O0` to `-O3` and even with `-ffast-math`** (fingerprint `0cfd4dfe` each time). That is a fact about this machine, not a guarantee: a processor with fused multiply-add instructions, or a different compiler, is allowed to give different last digits, and I did not test one. The **float world** is different. Its drift differs by 0.3 mm and its heading by 0.03° to 0.05°, and the odometry heading in the free-running case is -42.90° against -43.12°, a difference of 0.22°. The float simulator accumulates each wheel's travel in 32 bits, 1 mm at a time, and loses the small increments, so its encoders count slightly wrong.

A simulator is not neutral about its number type. The simulator's world is therefore kept in double, as the truth, while the robot code stays float, because that is what the board runs.

10. What the chapter's numbers can and cannot be trusted for

What can: the *code* (that the controller stops, that the fault triggers, that the replay is exact), the *comparisons* (more noise means more bias; more polling resolution means less; reversing removes the ratchet), and the *reasoning* they support. What cannot: any absolute figure for a real robot. The 5 mm of noise, the 2 % dropouts, the 0.1 s motor lag, the 5 % weak motor and the 0.5 m/s at a duty of 255 are assumptions from this chapter, and the simulator does not have stiction, slip, a sagging battery or timing jitter. The first test on the floor will disagree with the simulator, and the disagreement is the interesting part: each difference is a missing line in the table of section 3.

Challenges

CHALLENGE 1: FILTER AT THE SEAM

The ratchet bias of section 6 (-17.4 mm at 20 mm of noise) is a property of the one-way controller. Without letting the robot reverse and **without editing** `drive.h`, remove it by wrapping the hardware in a class with the same four functions that averages the last N valid ranges. Report the mean error, the spread and the worst case for N of 1, 3, 5 and 9 over 200 seeds at 20 mm of noise and 5 % dropouts.

View solution

CHALLENGE 2: HOW COARSE CAN AN ENCODER BE?

With the weak right motor and no heading hold the robot turns to -43°. Run it with encoders of 20, 90, 360 and 1440 ticks per wheel turn (the simulator and the controller must agree) and report the heading and position error of the odometry against the simulator's truth. Say what the millimetres per tick tell you.

View solution

CHALLENGE 3: A SEED IS A BUG REPORT

On the roughest setting of section 6 (20 mm of noise, 30 % dropouts) run 1000 seeds. Count the faults, find the seed whose run ended closest to the wall without a fault, and show that running that seed again reproduces the run exactly (a fingerprint of the final position), while the next seed does not. Do the same for the first faulting seed.

View solution

QUICK REFERENCE

- **The seam.** Four functions: `range_m()` (negative = no echo), `read_ticks()`, `drive(vl, vr)`, `bumped()`. Five implementations here: pins, simulator, recorder, replayer, averaging wrapper. `drive.h` is byte-identical in the simulator and the sketch (MD5 `51f2295b…`).
- **Simulator tests.** 19 checks against hand-worked answers, all pass (straight run x = 0.68020 against 0.68000; arc (0.52500, 1.22497); 1 m = 1736 ticks). Dropout fraction checked across 12 seeds: 9.95 %.

- **The wrong wall.** Weak right motor, no heading hold: -43°, 788.5 mm of drift, and stopped exactly 0.250 m from the bottom wall. Odometry heading -43.12° against truth -43.08°.
- **P-only heading hold.** Drift 788.5, 318.8, 122.0, 36.5 mm at gains 0, 2, 6, 20: roughly 1/gain; a steady disturbance leaves a steady error.
- **Noise.** Mean stop error +5.8 mm (no noise), -1.4 (5 mm), -17.5 (20 mm); with reversing allowed +7.3, +8.1, +8.3: the bias was the one-way controller. Polling every 1000 us stops 75.9 mm short.
- **Dropouts.** 30 % no-echo: 44 of 200 runs hit the fault (243 of 1000 in challenge 3); the robot stands still, never hits the wall.
- **Replay.** Same code: fingerprint `ce1530d6` both times, 0 differing commands. Gain 1.5 to 1.2: 178 of 500 commands differ from 4.32 s. Replay *cannot* give the closed-loop outcome (fingerprints `865c632a` against `04fbafcb`).
- **Budget.** `robot_ch3.ino`: 7726 bytes flash (23 %), 413 bytes RAM (20 %). Cheap trig: 7454 (-272), but the cosine is wrong by 0.0129 at 43° and has the wrong sign by 86°.
- **Reproducibility.** Identical output from `-O0` to `-O3` and `-ffast-math` on this machine (fingerprint `0cfd4dfe`); a float world differs (odometry heading -42.90° against -43.12°). Own random generator, so a seed names one run everywhere.
- **Challenges.** Averaging at the seam: bias -17.4, -5.9, -2.5, -0.1 mm for N = 1, 3, 5, 9; odometry position error 7.1, 0.8, 0.3, 0.1 mm for 20, 90, 360, 1440 ticks per turn; seed 302 is the worst fault-free run (-24.1 mm) and reproduces exactly.
- **Not run:** anything on a board; every simulator parameter is an assumption.

Next: Programming for Robotics with Arduino C++ 4, Sensor Code and Filtering

Programming for Robotics with Arduino C++ 4 (planned; not yet written) takes the filter that challenge 1 bolted on at the seam and builds the real ones: median, moving-average and exponential filters in integer and float versions, in fixed-size buffers with no lists, and the encoder decoding with interrupts that this chapter's sketch left as a stub, with the interrupt-safe reads that `volatile` data needs.

Programming for Robotics with Arduino C++

Course 2 · Chapter 4 · Sensor Code and Filtering

Programming for Robotics with Arduino C++ 3 ended with two loose ends. Challenge 1 bolted an averaging wrapper on at the seam and removed the ratchet bias, with a warning that the average costs delay; and `robot_ch3.ino` reported zero for the wheel encoders because they need interrupts. This chapter builds both properly: the small filters a robot needs, written without lists or the heap, with the ways each one fails, and the encoder code that runs inside an interrupt, with the one bug that interrupts invite.

Programming for Robotics with Arduino C++ 2 gave the rule that governs everything here: the Uno has 2048 bytes of RAM and 32,256 of flash, so every buffer has a size the compiler knows, and each piece is measured in bytes and cycles, not guessed at.

WHAT WAS RUN, AND WHAT WAS NOT

Run on a PC (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): every experiment with an output under it, and the challenge solutions. The signals are *models* with assumed noise, spike and chatter figures, and the IR sensor curve is an assumption, not any data sheet's. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`): the encoder sketch, the one-filter-at-a-time sketch and `robot_ch4.ino`. Flash and RAM figures are the compiler's; instruction counts come from disassembling the compiled code with `avr-objdump`. Cycle counts add up the data-sheet cycles of each instruction; they were *not* measured on a chip. **No board, encoder or sensor was attached.** Nothing here was measured on hardware.

1. No lists: ring buffers of fixed size

Python's `collections.deque(maxlen=5)` is a list that forgets its oldest entry, and it grows and shrinks as it likes. A filter on a microcontroller needs the same behaviour in memory that is reserved once. The standard answer is a **ring buffer**: an array of N values, a write position that wraps from the end back to the start, and a count of how many values are valid. Pushing overwrites the oldest value once the array is full. The array's size is a template parameter, so its RAM cost is known at compile time: `Ring<int16_t, N>` costs 2N + 2 bytes (the values, a head and a count).

IN PYTHON	ON THE UNO IN C++

`readings = deque(maxlen=5)`	`Ring<int16_t, 5> readings;` 12 bytes, reserved at compile time, never grows
`statistics.median(readings)`	a copy of at most N values, sorted by insertion sort: no allocation, and N is tiny
`sum(readings) / len(readings)`	a running `int32_t` sum, or a loop over the buffer; the float version has a trap (section 4)
a variable another thread changes: the interpreter protects a single assignment for you	a `volatile` variable an interrupt changes, read with interrupts off (section 9)
an integer never overflows or loses precision	shifts throw away bits (section 5); a `float` has 24 bits of precision (section 4)

Here is everything this chapter needs from the filters, in one header. Each class takes the numbers it needs and nothing else; none calls an Arduino function, so the same file runs in the PC experiments and compiles into the sketch.

```
// filters.h -- fixed-size buffers and the small filters a robot needs (Programming for
// Plain C++11: no Arduino calls, no heap, no lists. Every buffer's size is a template p
#pragma once
#include <stdint.h>

// ---- a ring buffer: the newest N values, oldest overwritten ----
template <class T, uint8_t N>
class Ring {
  static_assert(N > 0, "a ring buffer needs a size of at least 1");
 public:
  Ring() : head_(0), count_(0) {}
  void push(T v) { buf_[head_] = v; head_ = (uint8_t)((head_ + 1) % N); if (count_ < N)
  uint8_t size() const { return count_; }
  bool full() const { return count_ == N; }
  T at(uint8_t i) const { return buf_[(uint8_t)((head_ + N - count_ + i) % N)]; }   // i
  T oldest() const { return at(0); }
  T newest() const { return at((uint8_t)(count_ - 1)); }
 private:
  T buf_[N];
  uint8_t head_, count_;
};

// ---- debounce: the output changes only after the input has disagreed with it for n sa
class Debounce {
 public:
```

```
  explicit Debounce(uint8_t n, bool initial = false) : n_(n), count_(0), state_(initial)
  bool update(bool raw) {
    if (raw == state_) count_ = 0;
    else if (++count_ >= n_) { state_ = raw; count_ = 0; }
    return state_;
  }
  bool state() const { return state_; }
 private:
  uint8_t n_, count_;
  bool state_;
};

// ---- median of the last N values (N odd): rejects spikes completely, with no averagin
template <class T, uint8_t N>
class Median {
  static_assert(N % 2 == 1, "a median window must be odd");
 public:
  T update(T x) {
    ring_.push(x);
    T tmp[N];
    uint8_t n = ring_.size();
    for (uint8_t i = 0; i < n; ++i) {                  // insertion sort of a COPY, at m
      T v = ring_.at(i);
      uint8_t j = i;
      while (j > 0 && tmp[j - 1] > v) { tmp[j] = tmp[j - 1]; --j; }
      tmp[j] = v;
    }
    return tmp[n / 2];
  }
 private:
  Ring<T, N> ring_;
};

// ---- moving average of the last N values ----
template <uint8_t N>
class MovingAvgInt {                                   // integer running sum: exact, fo
 public:
  MovingAvgInt() : sum_(0) {}
  int16_t update(int16_t x) {
    if (ring_.full()) sum_ -= ring_.oldest();
    ring_.push(x);
    sum_ += x;
    return (int16_t)(sum_ / ring_.size());
  }
 private:
```

```
  Ring<int16_t, N> ring_;
  int32_t sum_;
};

template <uint8_t N>
class MovingAvgFloatSum {                              // the same idea in floats: add t
 public:
  MovingAvgFloatSum() : sum_(0.0f) {}
  float update(float x) {
    if (ring_.full()) sum_ -= ring_.oldest();
    ring_.push(x);
    sum_ += x;
    return sum_ / ring_.size();
  }
 private:
  Ring<float, N> ring_;
  float sum_;
};

template <uint8_t N>
class MovingAvgFloatSafe {                             // the float version that cannot 
 public:
  float update(float x) {
    ring_.push(x);
    float sum = 0.0f;
    for (uint8_t i = 0; i < ring_.size(); ++i) sum += ring_.at(i);
    return sum / ring_.size();
  }
 private:
  Ring<float, N> ring_;
};

// ---- exponential filter: y += a (x - y) ----
class EmaFloat {
 public:
  explicit EmaFloat(float a) : a_(a), y_(0.0f), init_(false) {}
  float update(float x) { if (!init_) { y_ = x; init_ = true; } else y_ += a_ * (x - y_)
 private:
  float a_, y_;
  bool init_;
};

// a = 1 / 2^K in integers. THIS ONE LOOKS RIGHT AND STALLS: when |x - y| < 2^K the shif
template <uint8_t K>
class EmaIntNaive {
```

```
 public:
  EmaIntNaive() : y_(0), init_(false) {}
  int16_t update(int16_t x) { if (!init_) { y_ = x; init_ = true; } else y_ += (int16_t)
 private:
  int16_t y_;
  bool init_;
};

// The fix: keep K extra BITS of y. acc holds y x 2^K, so the rounding that the shift th
template <uint8_t K>
class EmaIntGuard {
 public:
  EmaIntGuard() : acc_(0), init_(false) {}
  int16_t update(int16_t x) {
    if (!init_) { acc_ = (int32_t)x * (1L << K); init_ = true; }
    else acc_ += (int32_t)x - (acc_ / (1L << K));       // a division, not a shift: well
    return (int16_t)((acc_ + (1L << (K - 1))) / (1L << K));    // round to nearest
  }
 private:
  int32_t acc_;
  bool init_;
};
```

2. Debounce: a switch that lies for a few milliseconds

A bumper switch is metal touching metal, and the contact flickers for a few milliseconds each time it closes or opens. A program that reads the pin every millisecond sees a press as several. The cure is to accept a change only after the pin has disagreed with the current state *n samples in a row*, which is the `Debounce` class above. The experiment gives every press and release 0 to 8 ms of random chatter (an assumed model of the switch) and reads once a millisecond:

```
// e_debounce.cpp -- a bumper switch that chatters when pressed and released: how many p
// MODEL: every real change of the switch is followed by 0 to 8 ms of random chatter (th
#include <cstdio>
#include <initializer_list>
#include "sim.h"                                    // the seeded Rng
#include "filters.h"

struct Result { int presses_seen; double latency_sum_ms; int latency_n; int missed; };
```

```
static Result run(uint8_t n, double press_ms, uint32_t seed) {
  Rng rng(seed);
  Debounce deb(n);
  Result r = {0, 0.0, 0, 0};
  const int PRESSES = 50;
  const uint32_t period = 400;                          // one press every 400 ms
  bool prev_out = false;
  uint32_t press_start = 0; bool waiting = false; bool seen_this = false;
  uint32_t total = (uint32_t)PRESSES * period;           // exactly 50 press cycles
  for (uint32_t t = 0; t < total; ++t) {
    uint32_t phase = t % period;
    bool truth = phase >= 100 && phase < 100 + (uint32_t)press_ms;      // pressed from 
    // chatter: for the first `bounce` ms after each real edge, the contact flickers
    static uint32_t last_edge, bounce; static bool last_truth;
    if (t == 0) { last_edge = 0; bounce = 0; last_truth = false; }
    if (truth != last_truth) { last_edge = t; bounce = (uint32_t)(rng.uniform() * 9.0); 
    bool pin = truth;
    if (t - last_edge < bounce) pin = rng.uniform() < 0.5;
    if (phase == 100) { press_start = t; waiting = true; seen_this = false; }
    bool out = deb.update(pin);
    if (out && !prev_out) {                             // a rising edge of the DEBOUNC
      ++r.presses_seen;
      if (waiting && !seen_this) { r.latency_sum_ms += (double)(t - press_start); ++r.la
    }
    prev_out = out;
  }
  r.missed = PRESSES - r.latency_n;                     // presses that were never seen
  return r;
}

int main() {
  std::printf("50 presses of a switch with 0..8 ms of chatter after every change; the pi
  std::printf("  press length   samples in a row   press events seen   (true: 50)   mean
  for (double len : {30.0, 4.0})
    for (uint8_t n : {1, 3, 5, 10, 20}) {
      Result r = run(n, len, 1);
      std::printf("  %9.0f ms   %16u   %17d   %10s   %13.1f   %18d\n", len, (unsigned)n,
    }
  return 0;
}
```

```
50 presses of a switch with 0..8 ms of chatter after every change; the pin is read once 
  press length   samples in a row   press events seen   (true: 50)   mean delay ms   pre
         30 ms                 1                 129                          0.6
```

30 ms	3	52	4.8
30 ms	5	50	7.0
30 ms	10	50	12.0
30 ms	20	50	22.0
4 ms	1	103	0.7
4 ms	3	32	3.6
4 ms	5	12	4.6
4 ms	10	0	0.0
4 ms	20	0	0.0

Read the table from both ends. With no debounce (1 sample), a 30 ms press produced 129 press events for 50 true presses. Three samples in a row still gave 52, because eight milliseconds of chatter can include three equal readings in a row by chance; five samples gave exactly 50, at a mean delay of 7.0 ms, and 20 gave 50 at 22.0 ms. So the debounce time must be longer than the longest chatter, and every extra sample is delay added to *every* real press.

The lower half is the trap. A 4 ms tap is shorter than the chatter, so with 5 samples in a row only 12 of the 50 taps produced an event and 38 were never seen, and with 10 samples **all 50 were missed**. Debounce is a low-pass filter on a logic signal: it removes anything shorter than its window, whether that is chatter or a real, brief press. Choose the window from the shortest press that matters, not from the worst chatter alone; a bumper that must catch a brush against an obstacle needs a short window and a switch that chatters less.

3. Median or average? It depends on what the errors look like

An ultrasonic sensor makes two kinds of mistake: noise (a few millimetres either way), and **outliers**: no echo, or an echo from something off to the side, 3 m instead of 1 m. An average treats an outlier as data and smears it over the next N readings. A median sorts the last N values and takes the middle one, so one wrong value among N is simply discarded. The experiment feeds nine filters the same 200 seeded runs of a range falling from 2000 mm to 300 mm (5 mm noise, 5 % no-echo readings held, 3 % wrong echoes), and also a clean step from 1000 mm to 500 mm to measure delay:

```
// e_filters.cpp -- seven ways to filter an ultrasonic range, on a noisy signal with dro
// Signal: the true range falls from 2000 mm to 300 mm at 200 mm/s, one reading per 20 m
// 3 % wrong echoes (a random value 0..4000 mm). Dropouts do not reach the filter: it ho
#include <cstdio>
#include <cmath>
```

```
#include "filters.h"
#include "sim.h"

struct Acc { double sq = 0, worst = 0; long n = 0; };

template <class F>
static void run_noisy(F& f, Acc& a, uint32_t seed) {
  Rng rng(seed);
  int16_t out = 2000;
  const int N = 425;                                  // 8.5 s of 20 ms readings
  for (int i = 0; i < N; ++i) {
    double truth = 2000.0 - 200.0 * (i * 0.02);
    double u = rng.uniform();
    int16_t x;
    if (u < 0.05) x = -1;                             // no echo
    else if (u < 0.08) x = (int16_t)(rng.uniform() * 4000.0);   // a wrong echo
    else x = (int16_t)std::lround(truth + 5.0 * rng.normal());
    if (x >= 0) out = f.update(x);                    // dropouts are skipped: the ou
    if (i >= 20) { double e = out - truth; a.sq += e * e; if (std::fabs(e) > a.worst) a.w
  }
}

template <class F>
static int step_samples(F& f) {                       // clean step 1000 -> 500 mm: h
  for (int i = 0; i < 100; ++i) f.update(1000);
  for (int i = 1; i <= 200; ++i) { int16_t y = f.update(500); if (y <= 525) return i; }
  return -1;
}

struct Raw { int16_t update(int16_t x) { return x; } };
struct Cascade { Median<int16_t, 5> m; EmaIntGuard<2> e; int16_t update(int16_t x) { retu

template <class F>
static void row(const char* name) {
  Acc a;
  for (uint32_t seed = 1; seed <= 200; ++seed) { F f; run_noisy(f, a, seed); }
  F g;
  int s = step_samples(g);
  std::printf("  %-28s %9.1f   %9.1f   %14d   %13d\n", name, std::sqrt(a.sq / a.n), a.wo
}

int main() {
  std::printf("200 seeded runs of the falling range (5 mm noise, 5 %% no echo held, 3 %%
  std::printf("  filter                       rms mm      worst mm     readings to 500+2
  row<Raw>("none (hold on no echo)");
```

```
  row<Median<int16_t, 3> >("median of 3");
  row<Median<int16_t, 5> >("median of 5");
  row<Median<int16_t, 9> >("median of 9");
  row<MovingAvgInt<5> >("moving average of 5");
  row<MovingAvgInt<9> >("moving average of 9");
  row<EmaIntGuard<2> >("exponential, a = 1/4");
  row<EmaIntGuard<3> >("exponential, a = 1/8");
  row<Cascade>("median of 5, then a = 1/4");
  return 0;
}
```

```
200 seeded runs of the falling range (5 mm noise, 5 % no echo held, 3 % wrong echoes), a
  filter                       rms mm      worst mm     readings to 500+25   ms at 20 ms
  none (hold on no echo)          271.5      3545.0                1              20
  median of 3                      49.7      3277.0                2              40
  median of 5                      12.4      2155.0                3              60
  median of 9                      17.8        48.0                5             100
  moving average of 5             126.4      1800.0                5             100
  moving average of 9             101.0      1011.0                9             180
  exponential, a = 1/4            110.7      1486.0               11             220
  exponential, a = 1/8             91.0       903.0               23             460
  median of 5, then a = 1/4        22.4       561.0               13             260
```

With no filter the rms error is 271.5 mm, almost all of it from wrong echoes (worst 3545 mm). Averaging helps little: a moving average of 5 is still 126.4 mm rms and its worst error is 1800 mm, because one wrong echo of 3000 mm shifts the average by 600 mm for five readings. The median of 5 gets to 12.4 mm. The median of 9 has a better worst case (48 mm against 2155) but a worse rms (17.8 against 12.4): a bigger window lags the falling ramp more, and 200 mm/s is 4 mm per reading, so a median of N lags by about (N − 1)/2 readings.

The cost is in the last two columns: the median of 5 needs 3 readings (60 ms) to follow a step, the exponential filter with a = 1/8 needs 23 (460 ms), and the cascade (median of 5 to throw out spikes, then an exponential filter to smooth the noise) gave 22.4 mm rms for 13 readings of delay. **Smoothing and delay are the same thing**: a controller that acts on a 460 ms old range will overshoot a wall that is approaching, and this is why chapter 3's controller could use an average of 9 and a faster one could not.

A median is not magic either. It fails when *more than half* of its window is wrong; challenge 1 tabulates when that happens and compares it with the binomial chance.

4. The moving average that drifts

The fast way to a moving average is a running sum: add the new value, subtract the one that falls out, divide. In integers this is exact forever, and `MovingAvgInt` uses a 32-bit sum. In floats it looks identical and is not, because a `float` keeps about seven decimal digits, and every add and subtract rounds. The rounding errors do not cancel; they accumulate, in a sum that is never recomputed. The experiment runs a moving average of 8 on 10 million samples (at 50 Hz that is 2.3 days) for three input sizes, against a recompute-every-time float version and the exact answer:

```
// e_madrift.cpp -- a moving average of 8 values, kept as a running sum in float, agains
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "filters.h"

// The input is a value near `offset` that wobbles by +-3 in a fixed pattern. The exact
static double input(long i, double offset) { return offset + 3.0 * std::sin(0.01 * i) + (

static void trial(double offset, long checkpoints[], int ncp) {
  MovingAvgFloatSum<8> running;
  MovingAvgFloatSafe<8> safe;
  double window[8] = {0};
  int cp = 0;
  for (long i = 1; i <= checkpoints[ncp - 1]; ++i) {
    double x = input(i, offset);
    float fx = (float)x;                                  // what the board would hold
    float r = running.update(fx), s = safe.update(fx);
    window[i % 8] = fx;
    if (i == checkpoints[cp]) {
      double exact = 0; int n = i < 8 ? (int)i : 8;
      for (int k = 0; k < n; ++k) exact += window[((i - k) % 8 + 8) % 8];
      exact /= n;
      std::printf("  %9ld samples   running sum %12.4f (error %+9.4f)   recomputed %12.4f
      ++cp;
    }
  }
}

int main() {
  long cps[] = {1000, 100000, 1000000, 10000000};
  std::printf("input near 1000 (like a range in mm):\n");
```

```
  trial(1000.0, cps, 4);
  std::printf("input near 100000 (a larger offset, e.g. a scaled quantity):\n");
  trial(100000.0, cps, 4);
  std::printf("input near 1000000:\n");
  trial(1000000.0, cps, 4);
  // the integer version for comparison: exact by construction
  MovingAvgInt<8> iv;
  long long window[8] = {0};
  int16_t last = 0; long long worst = 0;
  for (long i = 1; i <= 10000000; ++i) {
    int16_t x = (int16_t)(1000 + (long)(3.0 * std::sin(0.01 * i)) + (i % 7));
    last = iv.update(x);
    window[i % 8] = x;
    if (i % 100000 == 0) {
      long long sum = 0; for (int k = 0; k < 8; ++k) sum += window[k];
      long long exact_floor = sum / 8;
      long long d = last > exact_floor ? last - exact_floor : exact_floor - last;
      if (d > worst) worst = d;
    }
  }
  std::printf("integer running sum (int32), 10,000,000 samples, checked every 100,000: t
  return 0;
}
```

```
input near 1000 (like a range in mm):
       1000 samples   running sum     998.7951 (error   +0.0002)   recomputed     998.79
     100000 samples   running sum    1002.7531 (error   +0.0087)   recomputed    1002.74
    1000000 samples   running sum     999.4631 (error   +0.0042)   recomputed     999.45
   10000000 samples   running sum    1000.6138 (error   +0.1017)   recomputed    1000.51
input near 100000 (a larger offset, e.g. a scaled quantity):
       1000 samples   running sum   99998.9062 (error   +0.1113)   recomputed   99998.79
     100000 samples   running sum  100002.3438 (error   -0.4023)   recomputed  100002.75
    1000000 samples   running sum   99993.8203 (error   -5.6377)   recomputed   99999.46
   10000000 samples   running sum   99946.7656 (error  -53.7461)   recomputed  100000.50
input near 1000000:
       1000 samples   running sum  999999.3750 (error   +0.5859)   recomputed  999998.87
     100000 samples   running sum 1000034.7500 (error  +32.0000)   recomputed 1000002.75
    1000000 samples   running sum 1000319.5625 (error +320.1016)   recomputed  999999.37
   10000000 samples   running sum 1003200.1250 (error +3199.6172)   recomputed 1000000.5
integer running sum (int32), 10,000,000 samples, checked every 100,000: the largest diff
```

For an input near 1000 the drift is small: 0.1017 after 10 million samples. For an input near 100000 the running sum is off by 53.7 and near 1,000,000 by 3199.6, while the recomputed version stays within 0.09 everywhere and the integer sum is exact (largest difference 0 over 10 million). The lesson is not "never use a float running sum"; it is that the error grows with the *size of the values* and the *length of the run*, and a robot that runs for days with an offset reading can hit both. Two safe options are the integer sum, or the float version that adds up the whole window each time (N additions, instead of two).

5. The exponential filter, and the integer one that stalls

The cheapest filter of all keeps no buffer: `y += a * (x - y)`, with a between 0 and 1. It needs one number of RAM. The integer version chooses a = $1/2^K$ so the multiply becomes a shift, and here is the trap: the shift throws bits away, and when the difference between input and output is smaller than 2^K the shifted difference is zero, so the output stops moving, short of the input. The experiment holds a constant input for 2000 samples and reports where each version ends:

```
// e_ema.cpp -- the exponential filter in float, in integers with a plain shift, and in
#include <cstdio>
#include <cmath>
#include "filters.h"
#include "sim.h"

template <class F>
static int settle(F& f, int16_t from, int16_t to, int steps) {   // feed `from` for a wh
  for (int i = 0; i < 400; ++i) f.update(from);
  int16_t y = from;
  for (int i = 0; i < steps; ++i) y = f.update(to);
  return y;
}

int main() {
  std::printf("final output after 2000 samples of a constant input (a step from 0, and a
  std::printf("  filter                          step 0 -> 1000   step 1000 -> 0   step 0
  { EmaIntNaive<2> a, b, c, d; std::printf("  integer, plain shift, K=2 (1/4) %14d   %15
  { EmaIntNaive<3> a, b, c, d; std::printf("  integer, plain shift, K=3 (1/8) %14d   %15
  { EmaIntNaive<5> a, b, c, d; std::printf("  integer, plain shift, K=5 (1/32)%14d   %15
  { EmaIntGuard<3> a, b, c, d; std::printf("  integer, guard bits,  K=3 (1/8) %14d   %15
  { EmaIntGuard<5> a, b, c, d; std::printf("  integer, guard bits,  K=5 (1/32)%14d   %15

  std::printf("\nagainst the float filter (a = 1/8) on a noisy ramp, 20000 samples of 5
```

```
  Rng rng(3);
  EmaFloat ref(0.125f);
  EmaIntNaive<3> naive;
  EmaIntGuard<3> guard;
  double sn = 0, sg = 0, mn = 0, mg = 0;
  int n = 0;
  for (int i = 0; i < 20000; ++i) {
    double truth = 3000.0 - 0.1 * i;                    // falls 0.1 mm per sample, so
    int16_t x = (int16_t)std::lround(truth + 5.0 * rng.normal());
    float r = ref.update((float)x);
    int16_t a = naive.update(x), b = guard.update(x);
    if (i > 100) { double dn = a - r, dg = b - r; sn += dn * dn; sg += dg * dg; mn += dn
  }
  std::printf("  plain shift   : mean difference from the float filter %+6.2f mm, rms %.
  std::printf("  guard bits    : mean difference from the float filter %+6.2f mm, rms %.
  return 0;
}
```

```
final output after 2000 samples of a constant input (a step from 0, and a step down from
  filter                         step 0 -> 1000   step 1000 -> 0   step 0 -> 37   step 0
  integer, plain shift, K=2 (1/4)            997                 0             34
  integer, plain shift, K=3 (1/8)            993                 0             30
  integer, plain shift, K=5 (1/32)           969                 0              6
  integer, guard bits,  K=3 (1/8)           1000                 1             37
  integer, guard bits,  K=5 (1/32)          1000                 1             37

against the float filter (a = 1/8) on a noisy ramp, 20000 samples of 5 mm noise round a
  plain shift   : mean difference from the float filter  -3.46 mm, rms 3.51 mm
  guard bits    : mean difference from the float filter  +0.50 mm, rms 0.58 mm
```

With the plain shift and K = 3, a step from 0 to 1000 ends at 993, which is $2^K - 1 = 7$ short; a step to 37 ends at 30; and a step from 0 to 5 **never moves off 0**, for K = 3 or 5. On the noisy ramp the plain shift reads an average of 3.46 mm low (about $(2^K - 1)/2$), against +0.50 mm for the version with guard bits. The fix is to keep K extra *bits* of the output: the accumulator holds the output times 2^K, so the rounding the shift would discard stays in the low bits and is fed back. The guard-bit version reaches 1000, 37 and 5. It is not perfect: falling to 0 ends at 1, a residue of the truncating division, and challenge 2 explores how many guard bits that costs to remove.

6. Calibration: from counts to distance and position

Every sensor reports a number, and a calibration turns it into the quantity the robot needs. Two examples show the two kinds of work involved. Both sensor models below are **assumptions made for these experiments**, not any manufacturer's curve.

An infrared distance sensor reports a curve that falls as an object recedes, roughly counts = 80 + 52000/(d + 40) from 100 to 800 mm, and **folds back** below 100 mm (the reading falls as the object gets too close). One function inverts the curve by formula (a float division, 900 bytes of flash in section 10); the other uses a table of N distances at evenly spaced counts and interpolates (integers only, but the table takes RAM). The line-sensor array has five sensors with different gain and offset, so its raw reading of the same line depends on which sensor sees it; the calibration sweeps over the line and floor, remembers each sensor's minimum and maximum, and scales every reading to 0 to 1000 before the weighted position is taken.

```
// calib.h -- turning raw sensor numbers into units (Programming for Robotics with Ardui
// The IR curve is an ASSUMPTION used for the experiments, not any datasheet's: counts(d
// and below 100 mm the reading FOLDS BACK (counts fall as the object gets closer), here
#pragma once
#include <stdint.h>

inline float ir_counts_model(float d_mm) {
  if (d_mm < 100.0f) return 451.0f * d_mm / 100.0f;
  return 80.0f + 52000.0f / (d_mm + 40.0f);
}

// Method 1: invert the formula. One float division. Only valid on the far side of the p
inline float ir_dist_formula(int16_t counts) {
  if (counts <= 80) return 9999.0f;                      // beyond the end of the curve:
  return 52000.0f / (float)(counts - 80) - 40.0f;
}

// Method 2: a table of distances at N evenly spaced counts from C_LO to C_HI, linear in
// table[i] is the distance (mm) for counts C_LO + i * STEP, nearest integer; counts out
template <uint8_t N>
class IrTable {
  static_assert(N >= 2, "a table needs at least two entries");
 public:
  IrTable(const int16_t* table, int16_t c_lo, int16_t c_hi) : t_(table), lo_(c_lo), span_
  int16_t dist(int16_t counts) const {
    int32_t pos = (int32_t)(counts - lo_) * (N - 1);     // position in table units, tim
    if (pos <= 0) return t_[0];
    if (pos >= (int32_t)span_ * (N - 1)) return t_[N - 1];
    int32_t i = pos / span_;                             // lower entry
```

```
    int32_t frac = pos - i * span_;                    // 0..span-1
    int32_t a = t_[i], b = t_[i + 1];
    return (int16_t)(a + ((b - a) * frac + (b - a >= 0 ? span_ / 2 : -span_ / 2)) / span_);
  }
 private:
  const int16_t* t_;
  int16_t lo_, span_;
};

// A line sensor array: five reflectance sensors, each with its own gain and offset. Min
// floor, remember each sensor's smallest and largest reading, then scale every reading t
template <uint8_t N>
class LineCal {
 public:
  LineCal() { for (uint8_t i = 0; i < N; ++i) { min_[i] = 32767; max_[i] = -32768; } }
  void learn(const int16_t* raw) {
    for (uint8_t i = 0; i < N; ++i) { if (raw[i] < min_[i]) min_[i] = raw[i]; if (raw[i]
  }
  int16_t scaled(uint8_t i, int16_t raw) const {
    int32_t range = (int32_t)max_[i] - min_[i];
    if (range < 1) return 0;
    int32_t v = ((int32_t)(raw - min_[i]) * 1000L) / range;
    if (v < 0) v = 0;
    if (v > 1000) v = 1000;
    return (int16_t)v;
  }
 private:
  int16_t min_[N], max_[N];
};

// Weighted position of a line under the array: sum(pos_i * v_i) / sum(v_i), in the units
template <uint8_t N>
inline bool line_position(const int16_t* v, const int16_t* pos_mm, float& out) {
  int32_t num = 0, den = 0;
  for (uint8_t i = 0; i < N; ++i) { num += (int32_t)pos_mm[i] * v[i]; den += v[i]; }
  if (den < 100) return false;
  out = (float)num / (float)den;
  return true;
}
```

```
// e_calib.cpp -- calibrating two sensors (host build; both sensor models are ASSUMPTIONS
// Part 1: an IR distance sensor. Its curve is counts = 80 + 52000/(d+40) from 100 to 80
// Part 2: a five-sensor line array with different gain and offset on every sensor: weigh
```

```
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <initializer_list>
#include "calib.h"
#include "sim.h"

static int16_t counts_at(float d) { return (int16_t)std::lround(ir_counts_model(d)); }

int main() {
  // ---- part 1a: how much distance does 2 counts of noise cost? ----
  std::printf("IR sensor (assumed curve): distance error caused by +2 counts of ADC nois
  std::printf("  true mm   counts   distance for counts+2   error mm   error %%\n");
  for (int d : {100, 200, 400, 800}) {
    int16_t c = counts_at((float)d);
    float est = ir_dist_formula((int16_t)(c + 2));
    std::printf("  %7d   %6d   %21.1f   %8.1f   %7.1f\n", d, c, est, est - d, 100.0 * (e
  }
  std::printf("  (and the same sensor at 50 mm reads %d counts: the SAME as it reads at
              (int)std::lround(ir_dist_formula(counts_at(50.0f))));

  // ---- part 1b: lookup table size vs interpolation error (noise-free, whole range) --
  std::printf("\nlookup table of N distances at evenly spaced counts (142..451), linear
  std::printf("  entries N   worst error mm   at counts   ints in table (bytes)\n");
  const int16_t C_LO = counts_at(800.0f), C_HI = counts_at(100.0f);
  const int Ns[] = {4, 8, 16, 32};
  for (int n : Ns) {
    static int16_t tab[64];
    for (int i = 0; i < n; ++i) {
      float c = C_LO + (float)(C_HI - C_LO) * i / (n - 1);
      tab[i] = (int16_t)std::lround(ir_dist_formula((int16_t)std::lround(c)));
    }
    double worst = 0; int wc = 0;
    for (int c = C_LO; c <= C_HI; ++c) {
      int16_t est;
      switch (n) {
        case 4:  { IrTable<4>  t(tab, C_LO, C_HI); est = t.dist((int16_t)c); } break;
        case 8:  { IrTable<8>  t(tab, C_LO, C_HI); est = t.dist((int16_t)c); } break;
        case 16: { IrTable<16> t(tab, C_LO, C_HI); est = t.dist((int16_t)c); } break;
        default: { IrTable<32> t(tab, C_LO, C_HI); est = t.dist((int16_t)c); } break;
      }
      double e = std::fabs(est - ir_dist_formula((int16_t)c));
      if (e > worst) { worst = e; wc = c; }
    }
    std::printf("  %9d   %14.1f   %9d   %21d\n", n, worst, wc, n * 2);
```

```
  }

  // ---- part 2: a line array ----
  const int16_t pos[5] = {-40, -20, 0, 20, 40};
  float gain[5] = {1.00f, 0.80f, 1.15f, 0.70f, 1.05f};
  float off[5]  = {  30,   90,   10,  120,   60};
  Rng rng(7);
  LineCal<5> cal;
  auto read = [&](float x, int16_t* raw) {               // x = the line's centre, mm; a
    for (int i = 0; i < 5; ++i) {
      float r = 1.0f - std::fabs(x - pos[i]) / 25.0f; if (r < 0) r = 0;
      raw[i] = (int16_t)std::lround(off[i] + gain[i] * 800.0f * r + 2.0 * rng.normal());
    }
  };
  int16_t raw[5];
  for (float x = -70; x <= 70; x += 1.0f) { read(x, raw); cal.learn(raw); }       // the
  std::printf("\nline array: position of a line from -30 to +30 mm, reading each sensor w
  std::printf("  true mm   uncalibrated   error   calibrated   error\n");
  double worst_raw = 0, worst_cal = 0, sq_raw = 0, sq_cal = 0; int n = 0;
  for (int x = -30; x <= 30; x += 5) {
    read((float)x, raw);
    int16_t scaled[5];
    for (int i = 0; i < 5; ++i) scaled[i] = cal.scaled((uint8_t)i, raw[i]);
    float pr = 0, pc = 0;
    line_position<5>(raw, pos, pr);
    line_position<5>(scaled, pos, pc);
    std::printf("  %7d   %12.1f   %5.1f   %10.1f   %5.1f\n", x, pr, pr - x, pc, pc - x);
  }
  for (int x = -30; x <= 30; ++x) {                      // every millimetre for the sta
    read((float)x, raw);
    int16_t scaled[5];
    for (int i = 0; i < 5; ++i) scaled[i] = cal.scaled((uint8_t)i, raw[i]);
    float pr = 0, pc = 0;
    line_position<5>(raw, pos, pr);
    line_position<5>(scaled, pos, pc);
    double er = pr - x, ec = pc - x;
    if (std::fabs(er) > worst_raw) worst_raw = std::fabs(er);
    if (std::fabs(ec) > worst_cal) worst_cal = std::fabs(ec);
    sq_raw += er * er; sq_cal += ec * ec; ++n;
  }
  std::printf("  over every mm from -30 to +30: worst error uncalibrated %.1f mm, calibra
              worst_raw, worst_cal, std::sqrt(sq_raw / n), std::sqrt(sq_cal / n));
  return 0;
}
```

```
IR sensor (assumed curve): distance error caused by +2 counts of ADC noise, formula inve
  true mm   counts   distance for counts+2   error mm   error %
      100      451                   99.4       -0.6      -0.6
      200      297                  197.4       -2.6      -1.3
      400      198                  393.3       -6.7      -1.7
      800      142                  772.5      -27.5      -3.4
  (and the same sensor at 50 mm reads 226 counts: the SAME as it reads at about 316 mm)

lookup table of N distances at evenly spaced counts (142..451), linear interpolation: wo
  entries N   worst error mm   at counts   ints in table (bytes)
          4            126.2         182                      8
          8             47.0         161                     16
         16             14.1         150                     32
         32              4.3         148                     64

line array: position of a line from -30 to +30 mm, reading each sensor with 2 counts of
  true mm   uncalibrated   error   calibrated   error
      -30          -21.4     8.6        -29.2     0.8
      -25          -18.8     6.2        -26.2    -1.2
      -20          -13.5     6.5        -19.5     0.5
      -15           -7.0     8.0        -12.8     2.2
      -10           -4.7     5.3         -9.7     0.3
       -5           -2.3     2.7         -6.1    -1.1
        0            1.0     1.0          0.1     0.1
        5            4.9    -0.1          6.5     1.5
       10            7.0    -3.0          9.6    -0.4
       15            9.6    -5.4         13.2    -1.8
       20           16.3    -3.7         19.8    -0.2
       25           22.1    -2.9         26.1     1.1
       30           25.2    -4.8         29.7    -0.3
  over every mm from -30 to +30: worst error uncalibrated 8.7 mm, calibrated 2.2 mm; rms
```

Three things stand out. First, a fixed error in counts is not a fixed error in millimetres: two counts of ADC noise cost 0.6 mm at 100 mm but 27.5 mm (3.4 %) at 800 mm, because the curve is flat there; a sensor is accurate near and vague far. Second, the fold-back means a reading does not always identify a distance: at 50 mm the sensor reads 226 counts, the same as it reads at about 316 mm, so a robot that can get closer than 100 mm must not trust the sensor alone. Third, the table's worst error shrinks as entries are added (126.2, 47.0, 14.1 and 4.3 mm for 4, 8, 16 and 32 entries), but at 16 entries it costs 32 bytes of RAM (section 10), and evenly spaced counts are not the best spacing for a curve this bent.

The line array shows what calibration is for. Without it the position estimate is off by up to 8.7 mm (rms 4.91 mm) over ±30 mm, purely because the sensors differ. With the minimum-and-maximum calibration the worst error is 2.2 mm (rms 0.90 mm), and what remains is mostly the noise. The calibration costs two numbers per sensor of RAM and a sweep before the run; a sensor that gets dirty or a floor that changes colour invalidates it, so the sweep belongs at the start of every run, not in the source code.

7. Quadrature encoders: three ways to count

A wheel encoder has two channels, A and B, that go high and low 90° apart. Moving forward, the pair of levels walks through the Gray sequence 00, 01, 11, 10, 00; moving backward it walks the other way. Each step is a **tick**, and the order tells the direction. The decoders below never read a pin: they are handed levels, so the same code runs on the PC and inside the interrupt.

```
// encoder.h -- decoding a quadrature encoder (channels A and B, 90 degrees apart). Plai
// Forward motion walks the two-bit state AB through the Gray sequence 00 -> 01 -> 11 ->
#pragma once
#include <stdint.h>

// Full 4x decoding: every edge of A or B counts, so 4 counts per cycle. Needs an interr
// A glitch that goes out and back (00 -> 01 -> 00 -> 01) counts +1 -1 +1 = +1: it canno
class Quad4x {
 public:
  Quad4x() : prev_(0), count_(0), errors_(0) {}
  void begin(uint8_t ab) { prev_ = ab & 3; }
  int8_t update(uint8_t ab) {                          // ab = (A << 1) | B; returns t
    ab &= 3;
    static const int8_t STEP[16] = {  0, +1, -1, 2,      // from 00
                                     -1,  0,  2, +1,     // from 01
                                     +1,  2,  0, -1,     // from 10
                                      2, -1, +1,  0 };   // from 11   (2 marks an impossi
    int8_t s = STEP[(prev_ << 2) | ab];
    prev_ = ab;
    if (s == 2) { ++errors_; return 0; }
    count_ += s;
    return s;
  }
  int32_t count() const { return count_; }
  uint32_t errors() const { return errors_; }
 private:
  uint8_t prev_;
```

```
  int32_t count_;
  uint32_t errors_;
};

// 2x decoding: an interrupt on the change of A ONLY; B is just read at that moment. Aft
// Half the resolution of 4x, but it needs one interrupt pin per wheel, which is what th
inline int8_t quad2x_step(bool a, bool b) { return (a == b) ? +1 : -1; }

// The simplest decoder, and the one most often written first: on a RISING edge of A, lo
class QuadNaive {
 public:
  QuadNaive() : prev_a_(false), count_(0) {}
  void update(bool a, bool b) {
    if (a && !prev_a_) count_ += b ? +1 : -1;      // forward: B is already high when A
    prev_a_ = a;
  }
  int32_t count() const { return count_; }
 private:
  bool prev_a_;
  int32_t count_;
};
```

`Quad4x` counts every edge of both channels (4 counts per cycle) and flags an impossible two-bit jump as an error. `quad2x_step` reacts only to a change of A and reads B at that moment (2 counts per cycle, and half the interrupts). `QuadNaive` counts the rising edges of A only (1 per cycle), the decoder most people write first. The test turns a simulated wheel forward at 0.5 m/s for 2 s and backward at 0.25 m/s for 1 s (1302 true ticks net), produces the A and B levels from the tick count, and corrupts them in two ways: *chatter* (an edge that goes out and back before settling) and *skipped states* (a state the sampler never saw, which is what happens when an interrupt is late):

```
// e_quad.cpp -- three quadrature decoders fed the A/B signals of a simulated wheel, wit
// The wheel is the simulator's: 2 s forward at 0.5 m/s then 1 s backward at 0.25 m/s, 3
#include <cstdio>
#include <cmath>
#include <vector>
#include <initializer_list>
#include "sim.h"
#include "encoder.h"

static void ab_of(int32_t tick, bool& a, bool& b) {
```

```
  int idx = (int)(((tick % 4) + 4) % 4);                // 0:00  1:01  2:11  3:10
  a = (idx == 2 || idx == 3);
  b = (idx == 1 || idx == 2);
}

struct Out { int32_t ticks, c4, c2, cn; uint32_t err4; int skipped, changes; };

static Out run(double chatter, double miss, uint32_t seed) {
  RobotParams rp; rp.tau = 0.0; rp.pwm_quantise = false;
  SensorParams sp; Arena arena;
  SimHW w(rp, sp, arena, seed);
  Rng rng(seed + 100);
  Quad4x q4; QuadNaive qn;
  int32_t c2 = 0;
  bool pa = false, pb = false;                          // the previous sample of A and
  q4.begin(0);
  int32_t last = 0; int skipped = 0, changes = 0;
  for (int ms = 0; ms < 3000; ++ms) {
    w.drive(ms < 2000 ? 0.5f : -0.25f, ms < 2000 ? 0.5f : -0.25f);
    w.step(0.001);
    int32_t l, r; w.read_ticks(l, r);
    while (last != l) {                                 // each tick of the wheel chang
      int32_t prev_tick = last;
      last += (l > last) ? 1 : -1;
      bool skip = rng.uniform() < miss; ++changes; if (skip) ++skipped;
      bool na, nb, oa, ob;
      ab_of(last, na, nb); ab_of(prev_tick, oa, ob);
      std::vector<std::pair<bool, bool> > samples;
      if (!skip) {
        if (rng.uniform() < chatter) { samples.push_back(std::make_pair(na, nb)); sample
        samples.push_back(std::make_pair(na, nb));
      }
      for (size_t k = 0; k < samples.size(); ++k) {
        bool a = samples[k].first, b = samples[k].second;
        q4.update((uint8_t)((a << 1) | b));
        if (a != pa) c2 += quad2x_step(a, b);           // the 2x decoder runs only whe
        qn.update(a, b);
        pa = a; pb = b;
      }
    }
  }
  (void)pb;
  int32_t l, r; w.read_ticks(l, r);
  Out o; o.ticks = l; o.c4 = q4.count(); o.c2 = c2; o.cn = qn.count(); o.err4 = q4.error
  return o;
```

```
}

int main() {
  std::printf("a wheel turned forward 2 s then backward 1 s; true tick count and what ea
  std::printf("  chatter on an edge   states skipped (of changes)   true ticks   4x coun
  for (double ch : {0.0, 0.02, 0.10, 0.30}) {
    Out o = run(ch, 0.0, 1);
    std::printf("  %17.0f %%   %6d of %-5d%3s   %10d   %8d   %8d   %11d   %14u\n", ch * 
  }
  for (double miss : {0.001, 0.01, 0.05}) {
    Out o = run(0.0, miss, 1);
    std::printf("  %17.0f %%   %6d of %-5d%3s   %10d   %8d   %8d   %11d   %14u\n", 0.0, 
  }
  return 0;
}
```

```
a wheel turned forward 2 s then backward 1 s; true tick count and what each decoder coun
  chatter on an edge   states skipped (of changes)   true ticks   4x count   2x count   
                 0 %        0 of 2170             1302       1302        651          
                 2 %        0 of 2170             1302       1302        651          
                10 %        0 of 2170             1302       1302        651          
                30 %        0 of 2170             1302       1302        651          
                 0 %        4 of 2170             1302       1298        649          
                 0 %       18 of 2170             1302       1270        641          
                 0 %      105 of 2170             1302       1160        583          
```

Chatter is harmless to the 4x and 2x decoders (1302 and 651 in every row) because a glitch that goes out and back counts +1 −1 +1, which is +1. The naive decoder is not so lucky: it should count 325.5 and counts 325, 323, 323 and 333, because every extra rising edge of A is another count.

Missing states is the harder case. Missing 4 of 2170 states cost the 4x decoder 4 counts and raised 4 error flags; the 2x decoder, which cannot see errors at all, was off by 2; and the naive one by about 2. I first expected two lost counts per skipped state, since a skipped state is a jump of two steps, which the 4x decoder flags and counts as zero. That is right, but the deficit is not 2 per skip, because a skipped state going *backward* adds 2 instead of subtracting them. With a throw-away variant of the program that also counted the direction of each skip (it printed 3 of the 4, 17 of 18 and 88 of 105 skips as forward), the deficits of 4, 32 and 142 are exactly 2 × (forward skips − backward skips): 2 × (3 − 1), 2 × (17 − 1) and 2 × (88 − 17). The error flags are

fewer than the skips at 5 % (97 against 105), because two skipped states in a row are a jump of three steps, which looks like a legal step in the other direction:

```
#include <cstdio>
#include "encoder.h"
static void ab_of(int t, bool& a, bool& b) { int i = ((t % 4) + 4) % 4; a = (i == 2 || i
int main() {
  // forward 0..20 skipping tick 5, and tick 12
  Quad4x q; q.begin(0);
  int skipped[] = {5, 12};
  for (int t = 1; t <= 20; ++t) {
    bool sk = false; for (int s : skipped) if (s == t) sk = true;
    if (sk) continue;
    bool a, b; ab_of(t, a, b); q.update((uint8_t)((a << 1) | b));
  }
  std::printf("forward 20 ticks, 2 isolated states never seen: count %d (expected 20), e
  // skipping two consecutive states
  Quad4x p; p.begin(0);
  for (int t = 1; t <= 20; ++t) { if (t == 5 || t == 6) continue; bool a, b; ab_of(t, a,
  std::printf("two CONSECUTIVE states never seen: count %d, errors %u\n", (int)p.count()
  return 0;
}
```

```
forward 20 ticks, 2 isolated states never seen: count 16 (expected 20), errors 2
two CONSECUTIVE states never seen: count 16, errors 0
```

Twenty forward ticks with two isolated skips counted 16 with 2 error flags, and with two consecutive skips counted 16 with **0 flags**. So an error flag is evidence of a lost state, and its absence is not evidence that none was lost. What you can do about it is make skips rare (a fast interrupt) and compare the encoder count with something else, as chapter 5 does with odometry.

8. Interrupts on the Uno

The Uno can interrupt on exactly two pins, 2 and 3 (INT0 and INT1). Two wheels with 4x decoding would need four interrupt sources; with 2x decoding, channel A of each wheel goes on an interrupt pin and channel B on any ordinary pin read inside the routine. `pins.h` now has `PIN_ENC_L_B = 15` and `PIN_ENC_R_B = 16` (analog pins A1 and A2 used as digital inputs) and

`PIN_IR = 17` (A3), each checked by a `static_assert`. (Full 4x decoding of two wheels would need pin-change interrupts, which share one vector between several pins; it is not built here.)

The sketch below writes the interrupt registers directly instead of calling `attachInterrupt`, so the routine stays short. It also has two reading functions, deliberately `noinline`, so that the compiler gives each its own code that can be read in the disassembly: one reads a 32-bit counter plainly, the other inside an atomic block (section 9).

```
// s_isr.ino -- two wheel encoders on the Uno's two interrupt pins, 2x decoding, and two
// Channel A of each wheel is on an interrupt pin (INT0 = pin 2, INT1 = pin 3); channel
// Compiled for the Uno with arduino-cli; NOT run on a board (none attached). The instru
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include "pins.h"

volatile int32_t g_ticks_l = 0, g_ticks_r = 0;

// After an edge of A: A == B means forward, A != B means backward (the 2x rule from enc
ISR(INT0_vect) {
  bool a = (PIND & _BV(PD2)) != 0;
  bool b = (PINC & _BV(PC1)) != 0;
  g_ticks_l += (a == b) ? 1 : -1;
}
ISR(INT1_vect) {
  bool a = (PIND & _BV(PD3)) != 0;
  bool b = (PINC & _BV(PC2)) != 0;
  g_ticks_r += (a == b) ? 1 : -1;
}

// UNSAFE: four separate byte loads. An interrupt between two of them gives a torn value
__attribute__((noinline)) int32_t read_left_unsafe() { return g_ticks_l; }

// SAFE: interrupts are off while the four bytes are copied, and put back as they were f
__attribute__((noinline)) int32_t read_left_atomic() {
  int32_t v;
  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { v = g_ticks_l; }
  return v;
}

void setup() {
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP);
  pinMode(PIN_ENC_L_B, INPUT_PULLUP); pinMode(PIN_ENC_R_B, INPUT_PULLUP);
  EICRA = _BV(ISC00) | _BV(ISC10);                      // INT0 and INT1: an interrupt
```

```
  EIFR = _BV(INTF0) | _BV(INTF1);                         // clear anything pending
  EIMSK = _BV(INT0) | _BV(INT1);                          // switch both on
  Serial.begin(115200);
}

void loop() {
  Serial.println(read_left_unsafe());
  Serial.println(read_left_atomic());
  Serial.println((long)g_ticks_r);
  delay(500);
}
```

```
arduino-cli compile --fqbn arduino:avr:uno --warnings all s_isr
Sketch uses 2418 bytes (7%) of program storage space. Maximum is 32256 bytes.
Global variables use 196 bytes (9%) of dynamic memory, leaving 1852 bytes for local vari

avr-objdump -d, _Z16read_left_unsafev:
  3c6  lds r22, 0x0116 ; 0x800116 <__data_end>
  3ca  lds r23, 0x0117 ; 0x800117 <__data_end+0x1>
  3ce  lds r24, 0x0118 ; 0x800118 <__data_end+0x2>
  3d2  lds r25, 0x0119 ; 0x800119 <__data_end+0x3>
  3d6  ret

avr-objdump -d, _Z16read_left_atomicv:
  3ae  in r18, 0x3f ; 63
  3b0  cli
  3b2  lds r22, 0x0116 ; 0x800116 <__data_end>
  3b6  lds r23, 0x0117 ; 0x800117 <__data_end+0x1>
  3ba  lds r24, 0x0118 ; 0x800118 <__data_end+0x2>
  3be  lds r25, 0x0119 ; 0x800119 <__data_end+0x3>
  3c2  out 0x3f, r18 ; 63
  3c4  ret

ISR for INT0 (__vector_1), counted from the listing: 54 instructions, 87 cycles on the f
plus about 7 cycles before its first instruction (4 for the interrupt, 3 for the jmp in
  11 x push
  11 x pop
  6 x ldi
  4 x sts
  4 x lds
  3 x in
  3 x adc
  2 x eor
  1 x rjmp
```

```
    1 x reti
    1 x out
    1 x movw
    1 x lsr
    1 x cpse
    1 x bst
    1 x bld
    1 x andi
    1 x add
```

The routine for INT0 compiles to 54 instructions. Adding the cycles of each (data-sheet figures: `push`, `pop`, `lds` and `sts` are two cycles, `reti` four) gives 87 cycles on the forward path and 84 on the backward path, and with about 7 cycles of interrupt entry, roughly **94 cycles, 5.9 µs at 16 MHz**. Half of that is overhead: 11 pushes and 11 pops save and restore registers the routine uses to do a four-byte add, 44 cycles, before and after a two-line routine. That is the price of a 32-bit counter on an 8-bit chip; a 16-bit counter would need only half the loads and stores. One wheel at the simulator's forward speed makes 868 ticks per second, so this 2x routine runs 434 times a second (868 would be 4x), a processor load of 434 × 94 / 16,000,000 = 0.25 % per wheel. These are cycle counts from a listing, not measurements of the chip: the cost of the interrupt entry is the data sheet's, and nothing was timed on a board.

9. The torn read

The Uno's processor is 8-bit. Reading a 32-bit `volatile` counter takes four separate `lds` instructions, as the listing above shows for `read_left_unsafe`, and an interrupt can arrive *between* any two of them. If the interrupt adds 1 to the counter in that gap, the program has the low byte from before and the upper bytes from after. Most of the time that does not matter: the bytes agree. It matters when the add carries out of the bytes already read, as in 0x000000FF becoming 0x00000100. The byte-load model enumerates the cases and then repeats 20 million random reads:

```
// e_torn.cpp -- a torn read: an 8-bit processor reads a 32-bit counter one byte at a ti
// This is a MODEL of the byte loads (lowest byte first, as avr-gcc loads them); it is n
#include <cstdio>
#include <cstdint>
#include <initializer_list>
#include "sim.h"
```

```
// Read the counter as four byte loads, with `isr_after` loads done when the interrupt (
static uint32_t torn_read(uint32_t counter, int isr_after) {
  uint32_t out = 0;
  for (int b = 0; b < 4; ++b) {
    if (b == isr_after) counter += 1;                    // the interrupt runs between by
    out |= ((counter >> (8 * b)) & 0xFFu) << (8 * b);
  }
  return out;
}

int main() {
  std::printf("a 32-bit counter read as four byte loads; an interrupt adds 1 after the f
  std::printf("  counter before   interrupt after load   value read     true values (bef
  for (uint32_t v : {0x00000010u, 0x000000FFu, 0x0000FFFFu, 0x00FFFFFFu}) {
    for (int g = 1; g <= 3; ++g) {
      uint32_t r = torn_read(v, g);
      long e1 = (long)r - (long)v, e2 = (long)r - (long)(v + 1);
      long err = (e1 < 0 ? -e1 : e1) < (e2 < 0 ? -e2 : e2) ? e1 : e2;
      std::printf("  0x%08X       %17d   0x%08X   %10u / %-10u   %16ld\n", (unsigned)v, g
    }
  }

  // How often? An interrupt that lands during one of the four load instructions takes e
  // gap 1, 2, 3 or after the last load with equal chance (the model: 4 loads of 2 cycle
  // Count over 20 million random (counter value, edge position) pairs how many reads ar
  Rng rng(11);
  const long TRIALS = 20000000L;
  long torn = 0, hit = 0;
  for (long i = 0; i < TRIALS; ++i) {
    uint32_t v = rng.next();                          // all 32 bits (uniform() has only 24
    int g = (int)(rng.uniform() * 4.0) + 1;           // 1,2,3: the interrupt lands in
    if (g < 4) ++hit;
    uint32_t r = torn_read(v, g);
    if (r != v && r != v + 1) ++torn;
  }
  std::printf("\n%ld random reads with an interrupt inside the read: %ld landed between
              TRIALS, hit, torn, 100.0 * torn / TRIALS, 100.0 * torn / hit, 100.0 / 256)

  // Arithmetic estimate for the encoder: a read takes about 4 x 2 = 8 cycles at 16 MHz
  double edges_per_s = 434.0;                         // ISR calls per second for one
  double read_s = 8.0 / 16e6;
  double p_hit = edges_per_s * read_s;                // chance an edge falls inside
  std::printf("arithmetic: one read lasts %.2f us; at %.0f ISR calls/s the chance an int
  std::printf("            the simulated rate above says about %.2e of such reads are to
```

```
              (double)torn / (double)hit * 0.75, 1.0 / (p_hit * (double)torn / (double)h
  std::printf("           at 50 reads per second that is one torn read every %.1f hours
              1.0 / (p_hit * (double)torn / (double)hit * 0.75) / 50.0 / 3600.0);
  return 0;
}
```

```
a 32-bit counter read as four byte loads; an interrupt adds 1 after the first, second or
  counter before   interrupt after load   value read     true values (before/after)   er
  0x00000010                       1   0x00000010           16 / 17
  0x00000010                       2   0x00000010           16 / 17
  0x00000010                       3   0x00000010           16 / 17
  0x000000FF                       1   0x000001FF          255 / 256
  0x000000FF                       2   0x000000FF          255 / 256
  0x000000FF                       3   0x000000FF          255 / 256
  0x0000FFFF                       1   0x000100FF        65535 / 65536
  0x0000FFFF                       2   0x0001FFFF        65535 / 65536
  0x0000FFFF                       3   0x0000FFFF        65535 / 65536
  0x00FFFFFF                       1   0x010000FF     16777215 / 16777216
  0x00FFFFFF                       2   0x0100FFFF     16777215 / 16777216
  0x00FFFFFF                       3   0x01FFFFFF     16777215 / 16777216            16

20000000 random reads with an interrupt inside the read: 14999877 landed between loads,
arithmetic: one read lasts 0.50 us; at 434 ISR calls/s the chance an interrupt falls ins
            the simulated rate above says about 9.66e-04 of such reads are torn, so a to
            at 50 reads per second that is one torn read every 26.5 hours of driving (th
```

The enumeration shows the wrong values: the counter 255 (0x000000FF) read as 511 when the interrupt lands after the first byte; 65535 (0x0000FFFF) read as 131071 (0x0001FFFF) when it lands after the second byte; and the worst case, 0x00FFFFFF read as 0x01FFFFFF, 16,777,215 too high. The random run found 19,315 torn reads in 20 million (0.097 %), close to the 1/1024 expected from an interrupt landing in the first gap one time in four and a low byte of 0xFF one time in 256. The arithmetic at the end is a model, not a measurement: with 434 interrupts per second (one wheel at 2x) and a read lasting about 8 cycles (0.5 µs), an interrupt lands inside a given read with chance 2.2×10^{-4}, so a torn read of this one counter happens about once per 4.8 million reads, one in 26 hours at 50 reads per second. That is rare enough to pass every test on a bench and wrong often enough to wreck a run of a robot that drives for hours; the wheel is then “somewhere else” for one control cycle, and the odometry carries the error forever.

The fix is to turn interrupts off while the four bytes are copied and put the interrupt flag back *as it was found* (`ATOMIC_BLOCK(ATOMIC_RESTORESTATE)`). The disassembly above shows what it costs:

the atomic read is 8 instructions and 15 cycles against 5 and 12, and interrupts are off for about 10 of them (0.6 µs), a tenth of one interrupt routine, so a pending edge is delayed by far less than the routine takes. `robot_ch4.ino` copies *both* counters inside one atomic block, so the left and right values are from the same moment.

10. What it costs, and the sketch

Each filter was compiled alone into a sketch that reads A3 and prints the result (cast to `long`, so that printing a float does not add its own code to the filter's bill). The first row is the sketch without a filter; the last two columns are what each filter adds:

```
arduino-cli compile --fqbn arduino:avr:uno --warnings all -DVARIANT_x s_filters
(flash and RAM of the whole sketch; "base" only reads A3 and prints it, so the last two
```

variant	flash bytes	RAM bytes	adds flash	adds RAM
base (no filter)	1890	188	0	0
debounce, 5 samples	1970	191	80	3
median of 3	2180	196	290	8
median of 5	2180	200	290	12
moving average of 8, integers	2142	210	252	22
moving average of 8, float	3000	222	1110	34
exponential, integers, K = 2	2122	193	232	5
exponential, float, a = 1/4	2876	197	986	9
IR distance, formula (float)	2790	188	900	0
IR distance, 16-entry table	2368	226	478	38

Floating point is the expensive habit: the float moving average adds 1110 bytes of flash and 34 of RAM against 252 and 22 for the integer one, because the Uno has no floating-point hardware and every `float` operation calls a library routine; an exponential filter in floats costs 986 bytes, in integers 232. For the IR sensor the 16-entry table (478 bytes of flash) is cheaper in flash than the formula (900) but costs 38 bytes of RAM, 32 of them the table itself: on the Uno, a `const` array without `PROGMEM` is copied into RAM at start-up. A `PROGMEM` table would give the RAM back; it was not tried here. The median of 3 and of 5 compile to the same size here, and differ only in the 4 bytes of buffer.

Finally `robot_ch4.ino` is chapter 3's robot with the stubbed encoders replaced by the interrupt routines and the atomic read, and a `Median<int16_t, 5>` between the ultrasonic sensor and the controller. The filter sits *in* `set_range`, the seam, so `drive.h` does not change (still byte-

identical to the simulator's). A no-echo reading is passed on as a no-echo: it is a fault signal, not a distance to average.

```
// robot_ch4.ino -- Programming for Robotics with Arduino C++, chapter 4: chapter 3's rob
// and a median filter between the ultrasonic sensor and the controller. Compile-checked
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include "scheduler.h"
#include "conv.h"
#include "echo.h"
#include "pins.h"
#include "filters.h"
#include "drive.h"

const float V_MAX_MPS = 0.5f;                          // wheel speed at a duty of 255: a

// ---- encoders: an interrupt on every change of channel A, channel B read inside it (2x
volatile int32_t g_ticks_l = 0, g_ticks_r = 0;
ISR(INT0_vect) {
  bool a = (PIND & _BV(PD2)) != 0, b = (PINC & _BV(PC1)) != 0;
  g_ticks_l += (a == b) ? 1 : -1;
}
ISR(INT1_vect) {
  bool a = (PIND & _BV(PD3)) != 0, b = (PINC & _BV(PC2)) != 0;
  g_ticks_r += (a == b) ? 1 : -1;
}
static void encoders_begin() {
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP);
  pinMode(PIN_ENC_L_B, INPUT_PULLUP); pinMode(PIN_ENC_R_B, INPUT_PULLUP);
  EICRA = _BV(ISC00) | _BV(ISC10);                     // any change on INT0 and on INT1
  EIFR = _BV(INTF0) | _BV(INTF1);
  EIMSK = _BV(INT0) | _BV(INT1);
}

class PinHW {                                          // the four functions drive.h nee
 public:
  PinHW() : range_m_(-1.0f) {}
  float range_m() { return range_m_; }                 // negative until the first echo,
  void set_range(float m) {                            // the filter sits HERE, at the s
    if (m < 0.0f) { range_m_ = m; return; }            // a dropout is passed on as a dr
    int16_t mm = (int16_t)(m * 1000.0f + 0.5f);
    range_m_ = (float)median_.update(mm) * 0.001f;
  }
```

```
  void read_ticks(int32_t& l, int32_t& r) {
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { l = g_ticks_l; r = g_ticks_r; }     // both coun
  }
  bool bumped() { return digitalRead(PIN_BUMP) == LOW; }
  void drive(float vl, float vr) {
    digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
    digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
    analogWrite(PIN_ML_PWM, speed_to_pwm(vl >= 0.0f ? vl : -vl, V_MAX_MPS));
    analogWrite(PIN_MR_PWM, speed_to_pwm(vr >= 0.0f ? vr : -vr, V_MAX_MPS));
  }
 private:
  float range_m_;
  Median<int16_t, 5> median_;
};

static PinHW hw;
static DriveConfig make_config() {
  DriveConfig c;
  c.stop_m = 0.25f; c.k_range = 1.5f; c.vmax = V_MAX_MPS; c.k_heading = 2.0f;
  c.wheel_r = 0.033f; c.track = 0.15f; c.ticks_per_rev = 360; c.max_dropouts = 5;   // 3
  return c;
}
static DriveRobot<PinHW> robot(hw, make_config());
static Echo echo;
static Scheduler<5> sch;
static bool trig_level = false;

static void echo_start_task(void*, uint32_t) { echo.start(); }
static void report_task(void*, uint32_t now) {
  int32_t l, r; hw.read_ticks(l, r);
  Serial.print(now); Serial.print(F(" ms  range ")); Serial.print(robot.range(), 3);
  Serial.print(F(" m  ticks ")); Serial.print(l); Serial.print(' '); Serial.print(r);
  Serial.print(F("  fault ")); Serial.println((int)robot.fault());
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT);
  pinMode(PIN_ML_PWM, OUTPUT); pinMode(PIN_MR_PWM, OUTPUT);
  pinMode(PIN_TRIG, OUTPUT); pinMode(PIN_ECHO, INPUT);
  pinMode(PIN_BUMP, INPUT_PULLUP);
  encoders_begin();
  sch.add(DriveRobot<PinHW>::sense_task, &robot, 20);
  sch.add(DriveRobot<PinHW>::think_task, &robot, 20);
  sch.add(DriveRobot<PinHW>::act_task, &robot, 20);
```

```
  sch.add(echo_start_task, 0, 60);
  sch.add(report_task, 0, 500);
}

void loop() {
  bool before = trig_level;
  Echo::Status st = echo.update(micros(), digitalRead(PIN_ECHO), trig_level);
  if (trig_level != before) digitalWrite(PIN_TRIG, trig_level);
  if (st == Echo::READY) hw.set_range(echo_us_to_m(echo.width_us()));
  else if (st == Echo::TIMEOUT) hw.set_range(-1.0f);
  sch.run(millis());
}
```

It compiles for the Uno, with no warnings under `--warnings all`, to **8626 bytes of flash (26 %) and 433 bytes of RAM (21 %)**, against 7726 and 413 for chapter 3's sketch: 900 bytes more flash and 20 bytes more RAM, of which 12 are the median's buffer and 8 the two counters. It was *not run*. In particular the `ticks_per_rev = 360` assumes a 180-slot disc at 2x decoding, and the right wheel, which turns the opposite way on a real robot, would need its sign flipped: a wiring and configuration question for the chapter that drives with these counts.

11. What the numbers can and cannot be trusted for

What they show. The shapes: a median rejects spikes while an average smears them; delay grows with the window; a plain integer exponential filter stalls by $2^K - 1$; a float running sum drifts with value size and run length; 4x and 2x decoders ignore chatter and the naive one does not; a 32-bit read on an 8-bit chip can tear, and an atomic read prevents it, at a measured cost in instructions. The instruction, flash and RAM counts come from the real compiler for the real chip.

What they do not show. Every noise level, spike rate, chatter length and curve here is an assumption, and a real sensor will have different ones, often with an error that is not random (a surface that absorbs sound, a switch that wears). The cycle counts are from the data sheet and a listing, not a logic analyser; the torn-read rate depends on how often the program really reads the counter, which was a guess of 50 Hz. And the decoders were tested on a perfect simulated wheel: real encoders add uneven slots and electrical noise that no experiment here has. Before a robot relies on any of this, put a scope or a logic analyser on the pins and read the tick counter against turns counted by hand.

CHALLENGE 1: HOW BIG A MEDIAN?

A median fails when more than half of its window is wrong. Sweep the share of wrong echoes (0, 3, 10, 20, 30 and 40 %) against windows of 3, 5, 9 and 15, on the falling-range signal of section 3 (100 seeds per cell), and print the rms and worst error. Next to it print the binomial chance that more than half of a window is wrong, and say what window you would choose for 3 % and for 20 % wrong echoes, and why the cleanest signal prefers the smallest window.

View solution

CHALLENGE 2: HOW MANY GUARD BITS?

Generalise the integer exponential filter so that the shift K and the number of guard bits G are separate (`EmaG<K, G>`). For K = 3 and K = 5 try several values of G and print where each ends after a step 0 → 1000, 0 → 37, 0 → 5 and 1000 → 0, and the rms difference from the float filter on a noisy ramp. Find the smallest G that removes the stall in both directions, and explain the result in terms of $(2^K - 1) / 2^G$.

View solution

CHALLENGE 3: WHICH VALUES TEAR?

Enumerate *every* value of a 2-byte and a 3-byte counter, with an interrupt adding 1 after each possible byte load, and count the torn reads exactly. Compare the counts with $1/256^{gap}$, report the worst error, and use the disassembly of the unsafe and atomic reads to say how many instructions and cycles the fix costs and for how long interrupts are off.

View solution

QUICK REFERENCE

- **Ring buffer.** `Ring<T, N>`: N values plus 2 bytes; no heap, no lists; the size is a template parameter.
- **Debounce.** 50 presses with 0–8 ms of chatter: 129 events raw, 52 at 3 samples in a row, 50 at 5 (delay 7.0 ms). A 4 ms tap: 12 of 50 seen at 5, none at 10. The window is a low-pass filter for real presses too.
- **Median or average.** rms 271.5 mm unfiltered, 126.4 averaging 5, 12.4 median of 5; worst 2155 mm for median 5, 48 for median 9. A window of N lags (N − 1)/2 readings; 460 ms for an exponential a = 1/8. A median fails when over half the window is wrong (challenge 1).

- **Float running sum.** Error after 10 million samples: 0.10 (input 1000), 53.7 (1e5), 3199.6 (1e6); recomputing the window stays within 0.09; an `int32_t` sum is exact.
- **Integer exponential filter.** Plain shift K=3 stops 7 short (993), never leaves 0 for a step to 5; guard bits fix it (1000, 37, 5), falling ends at 1; G = K + 1 clears both directions for K = 3 and 5 (challenge 2).
- **Calibration.** Two counts of noise cost 0.6 mm at 100 mm and 27.5 mm at 800 mm (assumed curve); a reading folds back below 100 mm (50 mm reads like about 316 mm). A 16-entry table: worst error 14.1 mm. Line array: worst 8.7 mm uncalibrated, 2.2 mm calibrated.
- **Decoders.** 4x = 1302, 2x = 651, naive = 325 on 1302 true ticks; chatter harmless to 4x/2x. Skipped states: deficit = 2 × (forward − backward skips); two consecutive skips raise no flag.
- **Interrupt.** INT0 routine: 54 instructions, about 94 cycles (5.9 µs) from a listing, 11 push and 11 pop of them; 0.25 % load per wheel at 434 calls/s.
- **Torn read.** A 32-bit read is 4 loads (5 instructions, 12 cycles); atomic: 8 instructions, 15 cycles, interrupts off for about 10. 19,315 of 20 million modelled reads torn (0.097 %); exhaustive 2-byte counter: 256 of 65,536 (1/256).
- **Budget.** `robot_ch4.ino`: 8626 bytes flash (26 %), 433 bytes RAM (21 %). Integer filters add 80–290 bytes of flash; floats 986–1110; a 16-entry table costs 32 bytes of RAM unless it is `PROGMEM`.
- **Not run:** anything on a board; every noise, spike and chatter figure and the IR curve are assumptions.

Next: Programming for Robotics with Arduino C++ 5, Driving: Motors and Odometry

Programming for Robotics with Arduino C++ 5 (planned; not yet written) turns the counters of this chapter into motion: motor driver code with ramping and limits, encoder counts converted to wheel speed, the odometry pose update, and straight and turn-by-angle moves, with the cost of `float` that this chapter measured and the encoder counts that its interrupt now supplies.

Programming for Robotics with Arduino C++

Course 2 · Chapter 5 · Driving: Motors and Odometry

Programming for Robotics with Arduino C++ 4 ended with a counter that is correct: an interrupt routine that counts each wheel's ticks, read atomically. This chapter uses those counts to make the robot *move*: a command that does not slam the motors, a wheel speed worked out from the ticks, a pose (x, y and heading) integrated from them, and two moves, straight and turn-by-angle, that end where they were told to. The theme is the one the course started with, the budget: what float costs on a chip with no floating-point hardware, and what the integer alternative costs in code and care.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): every experiment with an output under it and the challenge solutions, on the simulator of Programming for Robotics with Arduino C++ 3. The simulator's wheels have a 0.1 s lag, an 8-bit duty and exact ticks; the *dead zone*, the weak motor, and the wheel-radius and track errors used for calibration are **assumptions I chose**, not measurements of any motor. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`, no warnings from this chapter's code): the one-update-at-a-time sketch and `robot_ch5.ino`. Flash and RAM figures are the compiler's. **No time per update was measured**: nothing here was timed on a board. **No board, motor or encoder was attached.**

1. Ramps, limits and the dead zone

A controller that asks for 0.5 m/s at the instant it starts is asking a motor and a set of wheels for an acceleration they may not be able to give: the wheels slip, the supply sags, and the odometry (which assumes the wheels grip) goes wrong from the first centimetre. A **ramp** limits how fast a command may change. A **limit** clamps it. And a **dead zone** is the opposite problem at the other end: below some duty the motor does not turn at all, so a small command does nothing, and the robot sits still and then lurches. Here is the code of the chapter in one header, with its first pieces: `Ramp`, and `speed_to_pwm_dz`, which maps a speed to duty counts from just above the dead zone up to 255.

```
// motion.h -- driving a two-wheel robot (Programming for Robotics with Arduino C++ 5). I
// Ramp, motor command with a dead zone, two ways to turn encoder counts into a speed, a
#pragma once
```

```
#include <stdint.h>
#include <math.h>
#include "conv.h"

inline float clampf(float v, float lo, float hi) { return v < lo ? lo : (v > hi ? hi : v

// ---- ramp: the command may change by at most `accel` units per second ----
class Ramp {
 public:
  explicit Ramp(float accel) : a_(accel), v_(0.0f) {}
  float step(float target, float dt) { float m = a_ * dt; v_ += clampf(target - v_, -m, m
  float value() const { return v_; }
  void reset(float v = 0.0f) { v_ = v; }
 private:
  float a_, v_;
};

// ---- a motor command that knows about the dead zone: duty `dead` is the highest that
inline uint8_t speed_to_pwm_dz(float v, float vmax, uint8_t dead) {
  if (v <= 0.0f) return 0;
  if (v >= vmax) return 255;
  return (uint8_t)((float)dead + 1.0f + (254.0f - (float)dead) * v / vmax + 0.5f);
}

// ---- wheel speed, method 1: ticks counted over a fixed window. Resolution = one tick
class SpeedWindow {
 public:
  SpeedWindow(float per_tick_m) : pt_(per_tick_m), last_ticks_(0), last_us_(0), v_(0.0f),
  float update(int32_t ticks, uint32_t now_us) {
    if (first_) { last_ticks_ = ticks; last_us_ = now_us; first_ = false; return 0.0f; }
    uint32_t dt = now_us - last_us_;
    if (dt == 0) return v_;
    v_ = (float)(ticks - last_ticks_) * pt_ / ((float)dt * 1.0e-6f);
    last_ticks_ = ticks; last_us_ = now_us;
    return v_;
  }
 private:
  float pt_; int32_t last_ticks_; uint32_t last_us_; float v_; bool first_;
};

// ---- wheel speed, method 2: the time between the last two ticks. Finer when slow; nee
class SpeedPeriod {
 public:
  SpeedPeriod(float per_tick_m, uint32_t timeout_us) : pt_(per_tick_m), timeout_(timeout_
  void on_tick(int8_t dir, uint32_t t_us) { t_prev_ = t_last_; t_last_ = t_us; dir_ = di
```

```
  float speed(uint32_t now_us) const {
    if (n_ < 2 || now_us - t_last_ > timeout_) return 0.0f;      // no recent tick: stop
    return (float)dir_ * pt_ / ((float)(t_last_ - t_prev_) * 1.0e-6f);
  }
 private:
  float pt_; uint32_t timeout_, t_prev_, t_last_; int8_t dir_; uint8_t n_;
};

// ---- the pose update from one step of the wheels: distance d travelled by the centre
struct Pose { float x, y, th; };
inline void pose_euler(Pose& p, float d, float dth)    { p.x += d * cosf(p.th); p.y += d
inline void pose_midpoint(Pose& p, float d, float dth) { float m = p.th + 0.5f * dth; p.
inline void pose_arc(Pose& p, float d, float dth) {
  if (fabsf(dth) < 1.0e-6f) { pose_midpoint(p, d, dth); return; }
  float r = d / dth;
  p.x += r * (sinf(p.th + dth) - sinf(p.th));
  p.y -= r * (cosf(p.th + dth) - cosf(p.th));
  p.th += dth;
}

// ---- straight and turn-by-angle moves, on odometry. HW needs read_ticks() and drive()
struct MoveConfig {
  float wheel_r, track;            // metres: the ROBOT'S belief, which calibration corr
  int ticks_per_rev;
  float vmax, accel;               // cruise speed (m/s) and the ramp on the forward spe
  float k_dist;                    // forward speed per metre still to go (1/s)
  float k_head;                    // heading hold during a straight move: turn rate per
  float wmax, alpha, k_turn, wmin; // turning: top rate (rad/s), ramp (rad/s^2), rate pe
  float tol_m, tol_rad;            // a move is done when this close
};

template <class HW>
class Mover {
 public:
  enum Mode : uint8_t { IDLE, STRAIGHT, TURN };
  Mover(HW& hw, const MoveConfig& c) : hw_(hw), c_(c), vr_(c.accel), wr_(c.alpha), mode_
                                       p_{0.0f, 0.0f, 0.0f}, trav_(0.0f), target_(0.0f),
  void straight(float dist_m) { mode_ = STRAIGHT; start_ = trav_; target_ = dist_m; th_r
  void turn(float rad) { mode_ = TURN; target_ = p_.th + rad; }
  bool done() const { return mode_ == IDLE; }
  void update(float dt) {
    int32_t l, r; hw_.read_ticks(l, r);
    if (first_) { tl_ = l; tr_ = r; first_ = false; }
    const float pt = 6.2831853f * c_.wheel_r / (float)c_.ticks_per_rev;
    float dl = (float)(l - tl_) * pt, dr = (float)(r - tr_) * pt;
```

```
    tl_ = l; tr_ = r;
    float d = 0.5f * (dl + dr), dth = (dr - dl) / c_.track;
    pose_midpoint(p_, d, dth);
    trav_ += d;
    float v = 0.0f, w = 0.0f;
    if (mode_ == STRAIGHT) {
      float rem = target_ - (trav_ - start_);
      if (fabsf(rem) < c_.tol_m && fabsf(vr_.value()) < 0.02f) { mode_ = IDLE; vr_.reset
      else {
        v = vr_.step(clampf(c_.k_dist * rem, -c_.vmax, c_.vmax), dt);
        w = -c_.k_head * (p_.th - th_ref_) * (fabsf(v) / c_.vmax);
      }
    } else if (mode_ == TURN) {
      float rem = target_ - p_.th;
      if (fabsf(rem) < c_.tol_rad && fabsf(wr_.value()) < 0.05f) { mode_ = IDLE; vr_.res
      else {
        float want = clampf(c_.k_turn * rem, -c_.wmax, c_.wmax);
        if (fabsf(rem) >= c_.tol_rad && fabsf(want) < c_.wmin) want = rem > 0 ? c_.wmin
        if (fabsf(rem) < c_.tol_rad) want = 0.0f;
        w = wr_.step(want, dt);
      }
    }
    hw_.drive(v - 0.5f * w * c_.track, v + 0.5f * w * c_.track);
  }
  const Pose& pose() const { return p_; }
  float travelled() const { return trav_; }
 private:
  HW& hw_;
  MoveConfig c_;
  Ramp vr_, wr_;
  Mode mode_;
  bool first_;
  int32_t tl_, tr_;
  Pose p_;
  float trav_, target_, start_, th_ref_;
};
```

The experiment first drives the simulated wheel through chapter 2's plain `speed_to_pwm` and through the dead-zone version, with an assumed dead duty of 40 (the motor turns only above it, and speed rises linearly to 0.5 m/s at 255). Then it compares a step command with the same command through a ramp:

```
// e_motor.cpp -- two things between the controller and the wheels: the dead zone and th
// Part 1: the motors do not turn at duty <= 40 (an ASSUMED dead zone); above it speed r
// Part 2: a command that steps 0 -> 0.5 m/s against the same command through a Ramp; th
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "motion.h"

int main() {
  std::printf("steady wheel speed after 2 s for a commanded speed, dead zone at duty 40
  std::printf("  commanded   plain: duty   actual   error %%     with dead zone: duty   
  for (double v : {0.005, 0.02, 0.05, 0.1, 0.2, 0.4}) {
    double got[2]; int duty[2];
    for (int k = 0; k < 2; ++k) {
      RobotParams rp; rp.dead_duty = 40; rp.tau = 0.05;
      SensorParams sp; Arena arena;
      SimHW w(rp, sp, arena, 1);
      // the simulator quantises through speed_to_pwm itself, so for the compensated com
      uint8_t d = k == 0 ? speed_to_pwm((float)v, 0.5f) : speed_to_pwm_dz((float)v, 0.5f
      duty[k] = d;
      w.drive((float)(d / 255.0 * 0.5), (float)(d / 255.0 * 0.5));
      for (int i = 0; i < 2000; ++i) w.step(0.001);
      got[k] = w.wheel_l();
    }
    std::printf("  %9.3f   %11d   %6.3f   %7.0f     %20d   %6.3f   %7.1f\n", v, duty[0],
  }

  std::printf("\na step of 0.5 m/s against a ramp (wheel lag 0.1 s):\n");
  std::printf("  command            peak wheel acceleration (m/s^2)   time to 95 %% of s
  for (double accel : {0.0, 2.0, 1.0, 0.5}) {
    RobotParams rp; rp.tau = 0.10; rp.pwm_quantise = false;
    SensorParams sp; Arena arena;
    SimHW w(rp, sp, arena, 1);
    Ramp ramp((float)accel);
    double prev = 0, peak = 0, t95 = -1, d95 = 0, x0 = w.x;
    for (int i = 1; i <= 4000; ++i) {
      float cmd = accel == 0.0 ? 0.5f : ramp.step(0.5f, 0.001f);
      w.drive(cmd, cmd);
      w.step(0.001);
      double a = (w.wheel_l() - prev) / 0.001; prev = w.wheel_l();
      if (a > peak) peak = a;
      if (t95 < 0 && w.wheel_l() >= 0.475) { t95 = i * 0.001; d95 = w.x - x0; }
    }
```

```
      if (accel == 0.0) std::printf("  step               ");
      else std::printf("  ramp %.1f m/s^2     ", accel);
      std::printf("%29.2f   %25.3f   %26.3f\n", peak, t95, d95);
    }
    return 0;
  }
```

```
steady wheel speed after 2 s for a commanded speed, dead zone at duty 40 (m/s); vmax 0.5
  commanded   plain: duty   actual   error %     with dead zone: duty   actual   error %
      0.005             3    0.000      -100                       43    0.007      39.5
      0.020            10    0.000      -100                       50    0.023      16.3
      0.050            26    0.000      -100                       62    0.051       2.3
      0.100            51    0.026       -74                       84    0.102       2.3
      0.200           102    0.144       -28                      127    0.202       1.2
      0.400           204    0.381        -5                      212    0.400      -0.0

a step of 0.5 m/s against a ramp (wheel lag 0.1 s):
  command            peak wheel acceleration (m/s^2)   time to 95 % of speed (s)   dista
  step                                         5.00                       0.299
  ramp 2.0 m/s^2                               1.84                       0.448
  ramp 1.0 m/s^2                               0.99                       0.637
  ramp 0.5 m/s^2                               0.50                       1.068
```

With the plain mapping nothing below 0.1 m/s works: 0.005, 0.02 and 0.05 m/s all produce a duty of 26 or less, inside the dead zone, so the wheel does not turn (error −100 %), and 0.1 m/s gives 0.026 m/s, 74 % short. The dead-zone mapping puts them within 2.3 % from 0.05 m/s up. At the bottom it is not perfect: 0.005 m/s reads 39.5 % high, because one duty step is (0.5 − 0) / (255 − 40) = 2.3 mm/s, so the slowest steady speed is about 2.3 mm/s and a command below that is rounded up to it. The 40 is an assumption; a real motor's dead zone depends on the supply voltage, the load and the gearbox, and has to be found by driving it (challenge 3).

The ramp table shows what a ramp costs and buys. A step to 0.5 m/s makes the wheel's own lag the only limit: the peak acceleration is 5.00 m/s² and 95 % of the speed is reached in 0.299 s. A ramp of 1.0 m/s² keeps the peak at 0.99 m/s² and takes 0.637 s; of 0.5 m/s² takes 1.068 s. Whether 5 m/s² makes the wheels slip depends on the floor and the wheels, which the simulator does not model; the ramp is how you stay below the limit once you have found it.

2. From counts to wheel speed

The counter says how far the wheel has gone; a speed controller needs how fast it is going. There are two ways to get a speed from ticks. **Counting over a window** takes the difference of the count over a fixed time: `SpeedWindow`. Its resolution is one tick per window, so with 0.576 mm per tick and a window of 20 ms the smallest step in speed it can see is 28.8 mm/s. **Timing the ticks** takes the time between the last two: `SpeedPeriod`. Its resolution is the clock's, but it needs a timestamp for every tick, taken in the interrupt routine, and it must decide what to say when no tick has come lately. The experiment holds the wheel at a constant true speed:

```
// e_speed.cpp -- turning encoder counts into a wheel speed, two ways, on a wheel turnin
// 360 ticks per turn, wheel radius 0.033 m: one tick is 0.576 mm. The simulated clock s
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "motion.h"

struct Err { double sq = 0; int n = 0, zeros = 0; void add(double e, bool zero) { sq += 

int main() {
  std::printf("true speed held constant for 3 s; rms error of the speed estimate in mm/s
  std::printf("  true m/s   ms between ticks   window 20 ms   zeros of 20 ms   window 10
  for (double v : {0.01, 0.02, 0.05, 0.1, 0.2, 0.5}) {
    RobotParams rp; rp.tau = 0.0; rp.pwm_quantise = false;
    SensorParams sp; Arena arena;
    SimHW w(rp, sp, arena, 1);
    const float pt = (float)(2.0 * SIM_PI * rp.wheel_r / rp.ticks_per_rev);
    SpeedWindow w20(pt), w100(pt);
    SpeedPeriod per(pt, 200000);
    Err e20, e100, ep;
    int32_t last_l = 0;
    const uint32_t dt_us = 20;
    w.drive((float)v, (float)v);
    for (uint32_t us = dt_us; us <= 3000000; us += dt_us) {
      w.step(dt_us * 1e-6);
      int32_t l, r; w.read_ticks(l, r);
      if (l != last_l) { per.on_tick(l > last_l ? 1 : -1, us); last_l = l; }
      if (us % 20000 == 0) {
        float s20 = w20.update(l, us);
        if (us > 1000000) e20.add((s20 - v) * 1000.0, s20 == 0.0f);
        float sp_ = per.speed(us);
        if (us > 1000000) ep.add((sp_ - v) * 1000.0, false);
      }
      if (us % 100000 == 0) { float s100 = w100.update(l, us); if (us > 1000000) e100.ad
```

```
    }
    std::printf("  %8.2f   %16.1f   %12.2f   %14d   %13.2f   %19.2f\n", v, pt / v * 1000
  }
  return 0;
}
```

```
true speed held constant for 3 s; rms error of the speed estimate in mm/s (and how often
  true m/s   ms between ticks   window 20 ms   zeros of 20 ms   window 100 ms   period o
      0.01              57.6          13.74               65            2.50
      0.02              28.8          13.20               30            2.88
      0.05              11.5          12.63                0            2.64
      0.10               5.8          14.37                0            2.75
      0.20               2.9           6.84                0            2.64
      0.50               1.2          13.82                0            2.30
```

At 0.01 m/s a tick comes every 57.6 ms, so a 20 ms window sees no tick at all 65 times in 100 and one tick the rest, and the estimate flips between 0 and 28.8 mm/s: an rms error of 13.74 mm/s on a speed of 10. The period method is exact there (error 0.00). At 0.5 m/s the picture reverses a little: ticks are 1.2 ms apart, the 20 µs clock of this simulation is 1.7 % of that, and the period method's error is 4.36 mm/s (0.9 %), against 13.82 for the window. (The simulation's clock step is coarser than the Uno's 4 µs `micros()`, so this overstates the period method's error on the board.) The price of a longer window is delay: a window of 100 ms reads 2.5 mm/s rms error at 0.01 m/s, but its value is on average 50 ms old, which matters to a speed controller. Challenge 1 maps all of this.

3. Odometry: from ticks to a pose

Each control step the code turns the tick differences of the two wheels into the distance d moved by the centre and the heading change Δθ, and adds them to the pose. The question is which heading to move along while the heading is itself changing. Three answers, in increasing cost: **Euler** uses the heading at the start of the step; **midpoint** uses the heading half way through (as `drive.h` does); the **exact arc** uses the closed form for a circular arc. A fourth, in section 5, uses integers. The test drives two laps of a tight arc (radius 0.225 m, 1.67 rad/s) at several update intervals and compares the final pose with the simulator's truth:

```
// e_odom.cpp -- the same encoder counts through four odometry updates, on a tight arc (
// so what is left is the error of the update rule itself and of the tick quantisation).
```

```
// Path: left wheel 0.25 m/s, right wheel 0.5 m/s: an arc of radius 0.225 m at 1.667 rad/
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "motion.h"
#include "fixed.h"

struct R { double pos_mm, head_deg; };

template <int METHOD>
static R run(int interval_ms) {
  RobotParams rp; rp.tau = 0.0; rp.pwm_quantise = false;
  SensorParams sp; Arena arena;
  SimHW w(rp, sp, arena, 1);
  const float pt = (float)(2.0 * SIM_PI * rp.wheel_r / rp.ticks_per_rev);
  Pose p = {0.0f, 0.0f, 0.0f};
  OdomFixed fx((int32_t)std::lround(pt * 1e6 / 2.0 * 256.0), (int32_t)std::lround(pt / r
  int32_t pl = 0, pr = 0;
  fx.update(0, 0);                                    // the baseline: the counts at t
  w.drive(0.25f, 0.5f);
  const int total_ms = 7600;                          // a multiple of every update int
  for (int ms = 1; ms <= total_ms; ++ms) {
    for (int k = 0; k < 10; ++k) w.step(0.0001);
    if (ms % interval_ms == 0) {
      int32_t l, r; w.read_ticks(l, r);
      if (METHOD == 3) fx.update(l, r);
      else {
        float dl = (float)(l - pl) * pt, dr = (float)(r - pr) * pt;
        float d = 0.5f * (dl + dr), dth = (dr - dl) / (float)rp.track;
        if (METHOD == 0) pose_euler(p, d, dth); else if (METHOD == 1) pose_midpoint(p, d
      }
      pl = l; pr = r;
    }
  }
  double ox, oy, oth;
  if (METHOD == 3) { ox = fx.p.x_um * 1e-6; oy = fx.p.y_um * 1e-6; oth = fx.p.th * 2.0 *
  else { ox = p.x; oy = p.y; oth = p.th; }
  double ex = ox - (w.x - 0.3), ey = oy - (w.y - 1.0);
  double eth = std::remainder(oth - w.th, 2.0 * SIM_PI);
  R r; r.pos_mm = std::sqrt(ex * ex + ey * ey) * 1000.0; r.head_deg = eth * 180.0 / SIM_
  return r;
}

int main() {
```

```
  std::printf("two laps of a radius-0.225 m arc: position error (mm) / heading error (de
  std::printf("  interval   Euler (start heading)   midpoint (half heading)   exact arc
  for (int ms : {10, 20, 50, 100, 200}) {
    R a = run<0>(ms), b = run<1>(ms), c = run<2>(ms), d = run<3>(ms);
    std::printf("  %5d ms   %9.1f / %6.2f      %10.2f / %6.2f      %8.2f / %6.2f   %10.2
  }
  return 0;
}
```

```
two laps of a radius-0.225 m arc: position error (mm) / heading error (degrees) of the o
  interval   Euler (start heading)   midpoint (half heading)   exact arc          fixed
     10 ms         0.5 /   0.03           0.45 /   0.03          0.45 /   0.03
     20 ms         0.6 /   0.03           0.45 /   0.03          0.45 /   0.03
     50 ms         1.0 /   0.03           0.46 /   0.03          0.47 /   0.03
    100 ms         2.2 /   0.03           0.36 /   0.03          0.37 /   0.03
    200 ms         3.9 /   0.03           0.16 /   0.03          0.16 /   0.03       18
```

The first thing to see is the floor: every method has 0.45 mm of position error at 10 ms and 0.03° of heading error at every interval, and these are the encoder's quantisation (a tick is 0.576 mm) rather than anything in the update rule. Euler is a little worse (0.5 mm at 10 ms) and degrades as the interval grows: 1.0 mm at 50 ms, 3.9 mm at 200 ms, while midpoint and exact arc stay under 0.5 mm. Midpoint and exact arc agree to within 0.02 mm at every interval, so the exact arc, which costs 318 more bytes of flash (section 5), buys nothing here. Choose midpoint. The conclusion has a limit: the heading came from exact tick counts, so the only error was in where along the arc the step was taken; a real robot's error is dominated by slip and by the calibration of section 6.

4. Why float heading drifts

The pose update adds a small number to a large one: `th += dth`. A `float` has 24 bits of precision, so the spacing between neighbouring floats grows with the number: about 0.0039 near 32768, and near 6 about 5×10^{-7}. Every addition is rounded to a multiple of that spacing, and when the step is the same every time the rounding is the *same* every time, so it does not average out. This experiment adds 0.02 rad (1 rad/s at 50 Hz) to a heading that is kept three ways and compares each with the truth, computed in double:

```
// e_float.cpp -- adding small heading changes to a float heading, for a long time (host
// A robot turning at 1 rad/s with an update every 20 ms adds dth = 0.02 rad each step. 
// Four ways to keep the heading: a float that grows forever, a float wrapped into (-pi,
#include <cstdio>
#include <cmath>
#include <cstdint>
#include <initializer_list>

int main() {
  const double TWO_PI = 6.283185307179586;
  const float dth = 0.02f;
  float grow = 0.0f, wrap = 0.0f;
  uint16_t bam = 0; const uint16_t dbam = (uint16_t)std::lround(0.02 / TWO_PI * 65536.0)
  std::printf("heading after N updates of 0.02 rad (1 rad/s at 50 Hz); error = estimate 
  std::printf("  updates   time       float, never wrapped   float, wrapped to +-pi   ui
  long n = 0;
  for (long target : {1000L, 10000L, 100000L, 1000000L, 10000000L}) {
    for (; n < target; ++n) {
      grow += dth;
      wrap += dth; if (wrap > 3.14159265f) wrap -= 6.2831853f;
      bam = (uint16_t)(bam + dbam);
    }
    double truth = std::fmod(0.02 * (double)target, TWO_PI);
    auto err = [&](double est) { double e = std::remainder(est - truth, TWO_PI); return 
    double t_s = target / 50.0;
    std::printf("  %8ld   %5.1f h   %20.4f   %22.4f   %19.4f\n", target, t_s / 3600.0, e
  }
  std::printf("the 'never wrapped' heading itself ends at %.1f rad (truth %.1f rad)\n",
  std::printf("one unit of the binary angle is %.4f degrees; 209 units per step is %.6f 
  return 0;
}
```

```
heading after N updates of 0.02 rad (1 rad/s at 50 Hz); error = estimate - truth, in degr
  updates   time       float, never wrapped   float, wrapped to +-pi   uint16 binary angl
      1000     0.0 h                0.0153                  -0.0009                2.15
     10000     0.1 h                0.3383                  -0.0091               21.55
    100000     0.6 h               76.2079                  -0.0907             -144.43
   1000000     5.6 h               35.7660                  -0.9073               -4.30
  10000000    55.6 h               72.7898                  -9.0731              -43.01
the 'never wrapped' heading itself ends at 191361.9 rad (truth 200000.0 rad)
one unit of the binary angle is 0.0055 degrees; 209 units per step is 0.020038 rad again
```

A heading that is never wrapped is off by 76° after 100,000 updates (33 minutes of turning at 1 rad/s) and ends at 191,362 rad when the truth is 200,000: 4.3 % short, and the error is not random, it is rate error from rounding. Wrapping the angle into (−π, π] keeps the numbers small and cuts the error by a factor of about 800, to −0.09° after the same 100,000 updates, but it still drifts (−9.07° after 10 million). The third column, a `uint16_t` binary angle, looks worse (2.16° after 1000 updates, 21.6° after 10,000), and the reason is a lesson of its own: it adds 209 units per step, which is 0.02 rad rounded to the unit, a 0.19 % error in the *rate*, and that error accumulates exactly like the float's. Section 5 and challenge 2 show the cure: do not add steps, compute the heading from the exact integer tick count.

5. Odometry in integers

The Uno has no floating-point unit, so every `float` add, multiply and divide is a call to a library routine, and `sinf` and `cosf` are the heaviest of them. `fixed.h` does the update with integers only. The heading is a `uint16_t` in *binary angle units* (65,536 to the turn): it wraps by itself, with no wrap code, and its resolution is 0.0055°. The position is in micrometres in an `int32_t`. The sine comes from a 65-entry quarter-wave table in Q15 (15 fractional bits) with linear interpolation. The design point worth remembering is the one from section 4: the heading is computed from the *total* difference of the tick counts, so rounding in the conversion cannot accumulate; the distance keeps its rounding remainder and carries it into the next step.

```
// fixed.h -- the odometry update with integers only (Programming for Robotics with Ardu
// Heading is a uint16_t in binary angle units (65536 per turn): it wraps by itself, so
// Position is in micrometres (int32_t: good to +-2147 m). sin/cos come from a 65-entry
#pragma once
#include <stdint.h>

// sin(pi/2 * i/64) * 32767, i = 0..64
static const int16_t SIN_Q15[65] = {
      0,   804,  1608,  2410,  3212,  4011,  4808,  5602,  6393,  7179,  7962,  8739,  9
  12539, 13279, 14010, 14732, 15446, 16151, 16846, 17530, 18204, 18868, 19519, 20159, 20
  23170, 23731, 24279, 24811, 25329, 25832, 26319, 26790, 27245, 27683, 28105, 28510, 28
  30273, 30571, 30852, 31113, 31356, 31580, 31785, 31971, 32137, 32285, 32412, 32521, 32
  32767 };

inline int16_t sin_q15(uint16_t a) {                    // a: 0..65535 = 0..2 pi; result
  uint8_t quad = (uint8_t)(a >> 14);
  uint16_t r = a & 0x3FFF;
  if (quad & 1) r = (uint16_t)(0x4000 - r);              // mirror in the odd quadrants
```

```
  uint8_t i = (uint8_t)(r >> 8), frac = (uint8_t)r;
  int32_t lo = SIN_Q15[i], hi = SIN_Q15[i + 1 > 64 ? 64 : i + 1];
  int16_t v = (int16_t)(lo + (((hi - lo) * frac) >> 8));
  return (quad & 2) ? (int16_t)-v : v;
}
inline int16_t cos_q15(uint16_t a) { return sin_q15((uint16_t)(a + 0x4000)); }

struct PoseFixed { int32_t x_um, y_um; uint16_t th; };

// One step. dist_um = distance of the centre in micrometres, dth = heading change in bi
inline void pose_fixed(PoseFixed& p, int32_t dist_um, int16_t dth) {
  uint16_t mid = (uint16_t)(p.th + (uint16_t)(dth / 2));
  p.x_um += (dist_um * cos_q15(mid)) >> 15;             // 32-bit product: fine while |
  p.y_um += (dist_um * sin_q15(mid)) >> 15;
  p.th = (uint16_t)(p.th + (uint16_t)dth);
}

// Odometry from cumulative encoder counts, integers only. The heading is computed from
// so rounding in the conversion cannot accumulate; the distance keeps its rounding rema
//   k_dist_q8 = (micrometres per tick / 2) * 256      (the centre moves by half the sum
//   k_ang_q16 = (binary angle units per tick of difference) * 65536 = (per_tick / track
class OdomFixed {
 public:
  OdomFixed(int32_t k_dist_q8, int32_t k_ang_q16) : kd_(k_dist_q8), ka_(k_ang_q16), tl0_
  void update(int32_t tl, int32_t tr) {
    if (first_) { tl0_ = tl_ = tl; tr0_ = tr_ = tr; first_ = false; return; }
    int32_t dl = tl - tl_, dr = tr - tr_;
    tl_ = tl; tr_ = tr;
    int32_t q8 = (dl + dr) * kd_ + rem_;
    int32_t d_um = q8 >> 8;
    rem_ = q8 - (d_um << 8);
    uint32_t diff = (uint32_t)(tr - tr0_) - (uint32_t)(tl - tl0_);                  // t
    uint16_t th_new = (uint16_t)(th0_ + (uint16_t)((diff * (uint32_t)ka_) >> 16));   // 
    pose_fixed(p, d_um, (int16_t)(uint16_t)(th_new - p.th));
  }
  PoseFixed p;
  uint16_t th0_ = 0;                                    // initial heading, binary angle
 private:
  int32_t kd_, ka_, tl0_, tr0_, tl_, tr_, rem_;
  bool first_;
};
```

The column on the right of the odometry experiment is this code. Its heading error is the same 0.03° as the float versions and its position error 0.42 to 0.94 mm, the same as the float ones, with one exception that shows a limit of the design: at an interval of 200 ms the distance per step is 75 mm, over the 65 mm that a 32-bit product with a Q15 sine can take, and the error jumps to 185 mm. The code says so in its comment; the experiment shows what ignoring the comment costs. The cost side is measured by compiling one update at a time into a sketch that reads two counters and prints the pose (the flash and RAM of the whole sketch; the first row is the sketch with no odometry):

```
arduino-cli compile --fqbn arduino:avr:uno --warnings all -DVARIANT_x s_odom
(whole sketch; "base" only reads two counters and prints them; the last two columns are

  variant                                flash bytes   RAM bytes   adds flash   adds RAM
  base (no odometry)                            2134         196            0          0
  float, midpoint rule (sinf, cosf)             4032         216         1898         20
  float, exact arc (sinf, cosf)                 4350         216         2216         20
  integers, binary angle, 65-entry table        3206         367         1072        171
```

Midpoint odometry in float adds 1898 bytes of flash, the exact arc 2216 (the 318 extra are the division and the extra trigonometry), and the integer version 1072: a saving of 826 bytes, bought with 171 bytes of RAM, 130 of them the sine table. A `static const` array without `PROGMEM` is copied into RAM on the Uno, so the table is paid for twice (flash for its initial values and RAM for the copy); a `PROGMEM` table would give the RAM back and was not tried here. The integer version is not obviously a win on a 32 KB chip: 826 bytes is 2.6 % of the flash, and the code is harder to read. Its stronger case is time, which was *not measured*: the float version calls several library routines per update, and on a board with a faster control loop or less flash the integer version may be the only one that fits. Measure before choosing.

6. Moves: straight, and turn by an angle

`Mover` (in the header above) turns the pose into two moves. `straight(d)` drives until the odometry says d metres, with the forward speed proportional to the distance still to go (and ramped), and holds the heading by turning against the heading error (a P-only controller on the odometry heading). `turn(a)` turns until the heading is a radians away, with a turn rate proportional to the angle still to go, a floor on the rate (the slowest turn that is not stuck in the

dead zone), and a ramp. A move is *done* when it is within a tolerance and the speed has died away.

The first run of the turn never finished, and why is worth seeing:

```
// e_turn.cpp -- a turn-by-90-degrees move that never finishes (host build, ideal robot:
// The move is done when the heading is within a tolerance of the target and the turn ra
// 0.576 mm / 150 mm = 0.0038 rad (0.22 degrees); the slowest turn the controller allows
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "motion.h"

int main() {
  std::printf("turn +90 degrees, run for up to 10 s; what the TRUE heading does after th
  std::printf("  tolerance rad   tolerance deg   done at (s)   true heading at 10 s (deg
  for (double tol : {0.003, 0.006, 0.010, 0.020}) {
    RobotParams rp; rp.tau = 0.1;
    SensorParams sp; Arena big; big.w = 100; big.h = 100;
    SimHW w(rp, sp, big, 1); w.x = 50; w.y = 50;
    MoveConfig c;
    c.wheel_r = 0.033f; c.track = 0.15f; c.ticks_per_rev = 360; c.vmax = 0.4f; c.accel =
    c.wmax = 3.0f; c.alpha = 8.0f; c.k_turn = 3.0f; c.wmin = 0.4f; c.tol_m = 0.002f; c.t
    Mover<SimHW> m(w, c);
    m.turn((float)(SIM_PI / 2));
    double done_at = -1, lo = 1e9, hi = -1e9;
    for (int t = 0; t < 500; ++t) {
      m.update(0.02f);
      for (int k = 0; k < 20; ++k) w.step(0.001);
      if (done_at < 0 && m.done()) done_at = (t + 1) * 0.02;
      if (t >= 400) { double d = w.th * 180 / SIM_PI; if (d < lo) lo = d; if (d > hi) hi
    }
    char buf[16];
    if (done_at < 0) std::snprintf(buf, sizeof buf, "never"); else std::snprintf(buf, si
    std::printf("  %13.3f   %13.2f   %11s   %26.2f   %42.2f\n", tol, tol * 180 / SIM_PI,
  }
  return 0;
}
```

```
turn +90 degrees, run for up to 10 s; what the TRUE heading does after the controller de
  tolerance rad   tolerance deg   done at (s)   true heading at 10 s (deg)   true headin
          0.003            0.17         never                        89.02
```

```
          0.006           0.34         never                    89.02
          0.010           0.57          1.46                    88.47
          0.020           1.15          1.20                    92.06
```

With tolerances of 0.003 and 0.006 rad the 90° turn **never completes**: the true heading swings 1.78° peak to peak for as long as the program runs. The tolerance (0.17° and 0.34°) is smaller than one step of the heading the encoders can see (one tick on each wheel is 0.44°), and the controller will not turn slower than 0.4 rad/s, so every time it moves it jumps over the target, reverses, and jumps back. At 0.010 rad (0.57°) it finishes in 1.46 s. The rule is a general one: **a tolerance finer than the sensor's resolution is a request the controller can never satisfy**; set it at about two steps of resolution. The finished turn is still 1.5° short (88.47°); I did not separate how much of that is the tolerance, the quantisation and the last coasting of the wheels.

Now the calibration. A robot's code believes a wheel radius and a track width; the real ones differ, and an error in the radius is an error in every distance, and an error in the track is an error in every turn. Two driving tests measure them: drive a long straight and compare the distance the odometry counted with the distance covered; then spin on the spot for several turns and compare. In the experiment the true robot has a wheel radius 1.5 % larger than the code's, a track 1.3 % wider, and a right motor 5 % weak (all assumed):

```
// e_move.cpp -- straight and turn-by-angle moves on odometry, with a robot whose real wh
// The true robot: right motor 5 % weak, wheel radius 0.0335 m (the code believes 0.033),
#include <cstdio>
#include <cmath>
#include "sim.h"
#include "motion.h"

static RobotParams true_robot() {
  RobotParams rp; rp.wheel_r = 0.0335; rp.track = 0.152; rp.gain_r = 0.95; rp.tau = 0.1;
  return rp;
}
static MoveConfig config(double wheel_r, double track, float k_head) {
  MoveConfig c;
  c.wheel_r = (float)wheel_r; c.track = (float)track; c.ticks_per_rev = 360;
  c.vmax = 0.4f; c.accel = 1.0f; c.k_dist = 3.0f; c.k_head = k_head;
  c.wmax = 3.0f; c.alpha = 8.0f; c.k_turn = 3.0f; c.wmin = 0.4f;
  c.tol_m = 0.002f; c.tol_rad = 0.010f;           // 0.57 degrees: coarser than one tick
  return c;
}
```

```
struct Result { double dist, lateral, head_deg; };

// run one move on a fresh robot; returns what the TRUTH did
template <bool TURN>
static Result run(double amount, double wheel_r, double track, float k_head) {
  RobotParams rp = true_robot();
  SensorParams sp; Arena big; big.w = 100; big.h = 100;
  SimHW w(rp, sp, big, 1);
  w.x = 50.0; w.y = 50.0;
  MoveConfig c = config(wheel_r, track, k_head);
  Mover<SimHW> m(w, c);
  if (TURN) m.turn((float)amount); else m.straight((float)amount);
  for (int tick = 0; tick < 3000 && !m.done(); ++tick) { m.update(0.02f); for (int k = 0
  for (int k = 0; k < 600; ++k) { m.update(0.02f); for (int j = 0; j < 20; ++j) w.step(0
  Result r;
  double dx = w.x - 50.0, dy = w.y - 50.0;
  r.dist = std::sqrt(dx * dx + dy * dy);
  r.lateral = dy;                                    // the move was ordered along +x
  r.head_deg = w.th * 180.0 / SIM_PI;
  return r;
}

int main() {
  const double R0 = 0.033, T0 = 0.150;
  std::printf("before calibration (the code believes wheel radius 0.033 m, track 0.150 m
  std::printf("  move                                 true distance (m)   sideways drift
  Result a = run<false>(1.0, R0, T0, 0.0f);
  std::printf("  straight 1.000 m, no heading hold    %17.4f   %19.1f   %18.2f\n", a.dis
  Result b = run<false>(1.0, R0, T0, 3.0f);
  std::printf("  straight 1.000 m, heading hold       %17.4f   %19.1f   %18.2f\n", b.dis
  Result t = run<true>(SIM_PI / 2, R0, T0, 0.0f);
  std::printf("  turn +90 degrees                     %17s   %19s   %18.2f\n", "", "", t

  // calibration 1: drive 2 m by the code's belief and measure the true distance -> a sc
  Result s = run<false>(2.0, R0, T0, 3.0f);
  double scale_r = s.dist / 2.0;
  double R1 = R0 * scale_r;
  // calibration 2: spin four turns by the code's belief (with the radius corrected) and
  Result sp4 = run<true>(8.0 * SIM_PI, R1, T0, 0.0f);
  double true_turn = sp4.head_deg * SIM_PI / 180.0;
  double T1 = T0 * (8.0 * SIM_PI) / true_turn;
  std::printf("\ncalibration: a 2 m drive covered %.4f m, so the wheel radius becomes %.
              s.dist, R1, scale_r, true_turn / (2 * SIM_PI), sp4.head_deg, T1);
  std::printf("(the true values are 0.0335 and 0.152; the 5 %% weaker right motor does n
```

```
  std::printf("\nafter calibration:\n");
  std::printf("  move                                  true distance (m)   sideways drift
  Result a2 = run<false>(1.0, R1, T1, 0.0f);
  std::printf("  straight 1.000 m, no heading hold    %17.4f   %19.1f   %18.2f\n", a2.di
  Result b2 = run<false>(1.0, R1, T1, 3.0f);
  std::printf("  straight 1.000 m, heading hold       %17.4f   %19.1f   %18.2f\n", b2.di
  Result t2 = run<true>(SIM_PI / 2, R1, T1, 0.0f);
  std::printf("  turn +90 degrees                     %17s   %19s   %18.2f\n", "", "", t
  return 0;
}
```

```
before calibration (the code believes wheel radius 0.033 m, track 0.150 m):
  move                                  true distance (m)   sideways drift (mm)   true he
  straight 1.000 m, no heading hold                1.0103                -172.2
  straight 1.000 m, heading hold                   1.0152                 -42.1
  turn +90 degrees

calibration: a 2 m drive covered 2.0305 m, so the wheel radius becomes 0.03350 m (scale
(the true values are 0.0335 and 0.152; the 5 % weaker right motor does not change either

after calibration:
  move                                  true distance (m)   sideways drift (mm)   true he
  straight 1.000 m, no heading hold                0.9955                -167.2
  straight 1.000 m, heading hold                   1.0002                 -41.4
  turn +90 degrees
```

Before calibration an ordered 1 m straight covers 1.0152 m with the heading held: 1.5 % long, which is the radius. The 2 m calibration drive measured 2.0305 m, so the radius is corrected to 0.03350 m (the true value, to the digits shown). The four-turn spin gave 3.943 turns instead of 4, and the track is corrected to 0.15216 m (true 0.152; the 0.1 % residual is the turn tolerance). After calibration the held straight covers 1.0002 m. The 5 % weak right motor does not enter either number: both calibrations come from the *ticks*, which count what the wheels really did, whatever the duty. What the weak motor does is steer, and without heading hold the robot ends 167 mm to the side at −19.3° (and a chord of 0.9955 m for 1 m of path); with the P-only heading hold the drift falls to 41 mm and −2.6°. A proportional hold leaves a steady error against a steady disturbance; removing that is the job of the PID of Programming for Robotics with Arduino C++ 6.

The turn is the sobering row: +90° ordered gave 88.93° before and 88.78° after calibration. For turns the two errors nearly cancel (a radius 1.5 % too small and a track 1.3 % too narrow: a net

0.18 %, which is 0.16° at 90°), and calibration moved the result by 0.15°, as that predicts, which is far less than the 1.1° the stopping error leaves. The track alone is a larger error: with the radius corrected but not the track, the four ordered turns fell 20° short. Know the size of each error before calibrating the smallest one.

7. The sketch and the budget

`robot_ch5.ino` puts it together on the Uno: the interrupt routines and atomic reads of Programming for Robotics with Arduino C++ 4, `PinHW` reduced to `read_ticks` and `drive` (all that `Mover` needs), a motor command through `speed_to_pwm_dz` with an *assumed* dead duty of 40, and a control task every 20 ms that drives a square: 0.5 m, a 90° turn, four times. It prints the pose and the left wheel speed twice a second.

```
// robot_ch5.ino -- Programming for Robotics with Arduino C++, chapter 5: the robot drive
// Compile-checked for the Uno (arduino:avr:uno). NOT run on a board: no board, motor or
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include "scheduler.h"
#include "conv.h"
#include "pins.h"
#include "motion.h"

const float V_MAX_MPS = 0.5f;                          // wheel speed at a duty of 255:
const uint8_t DEAD_DUTY = 40;                          // the highest duty that does not

volatile int32_t g_ticks_l = 0, g_ticks_r = 0;         // the interrupt routines of chap
ISR(INT0_vect) { bool a = (PIND & _BV(PD2)) != 0, b = (PINC & _BV(PC1)) != 0; g_ticks_l
ISR(INT1_vect) { bool a = (PIND & _BV(PD3)) != 0, b = (PINC & _BV(PC2)) != 0; g_ticks_r

class PinHW {                                          // read_ticks() and drive(): all
 public:
  void read_ticks(int32_t& l, int32_t& r) { ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { l = g_ti
  void drive(float vl, float vr) {
    digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
    digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
    analogWrite(PIN_ML_PWM, speed_to_pwm_dz(vl >= 0.0f ? vl : -vl, V_MAX_MPS, DEAD_DUTY)
    analogWrite(PIN_MR_PWM, speed_to_pwm_dz(vr >= 0.0f ? vr : -vr, V_MAX_MPS, DEAD_DUTY)
  }
};
```

```
static PinHW hw;
static MoveConfig make_config() {
  MoveConfig c;
  c.wheel_r = 0.033f; c.track = 0.150f; c.ticks_per_rev = 360;       // 360 COUNTS per t
  c.vmax = 0.4f; c.accel = 1.0f; c.k_dist = 3.0f; c.k_head = 3.0f;
  c.wmax = 3.0f; c.alpha = 8.0f; c.k_turn = 3.0f; c.wmin = 0.4f;
  c.tol_m = 0.002f; c.tol_rad = 0.010f;                              // coarser than one
  return c;
}
static Mover<PinHW> mover(hw, make_config());
static SpeedWindow speed_l(2.0f * 3.14159265f * 0.033f / 360.0f);
static Scheduler<3> sch;
static uint8_t leg = 0;                                              // 0..7: straight,

static void control_task(void*, uint32_t) {
  mover.update(0.02f);
  if (mover.done() && leg < 8) {
    if (leg % 2 == 0) mover.straight(0.5f); else mover.turn(1.5707963f);
    ++leg;
  }
}
static void report_task(void*, uint32_t) {
  int32_t l, r; hw.read_ticks(l, r);
  Serial.print(F("leg ")); Serial.print(leg); Serial.print(F("  x ")); Serial.print(mover
  Serial.print(F("  y ")); Serial.print(mover.pose().y, 3); Serial.print(F("  th ")); Se
  Serial.print(F("  v_left ")); Serial.println(speed_l.update(l, micros()), 3);
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT); pinMode(PIN_ML_PWM, OUTPUT);
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP); pinMode(PIN_ENC_L_
  EICRA = _BV(ISC00) | _BV(ISC10); EIFR = _BV(INTF0) | _BV(INTF1); EIMSK = _BV(INT0) | _
  sch.add(control_task, 0, 20);
  sch.add(report_task, 0, 500);
}

void loop() { sch.run(millis()); }
```

It compiles with no warnings under `--warnings all` to **8868 bytes of flash (27 %) and 409 bytes of RAM (19 %)**, against 8626 and 433 for Programming for Robotics with Arduino C++ 4's sketch: it adds the odometry (with `sinf` and `cosf`) and the moves, and drops the ultrasonic sensor and its median filter. It was *not run*. The values that must change before it

moves a real robot are the ones the chapter has just shown to matter: the wheel radius and track from a calibration, `DEAD_DUTY` from the sweep of challenge 3, and `ticks_per_rev` from the encoder disc.

8. What the numbers can and cannot be trusted for

What they show. The shapes: a window cannot see a slow wheel and a period can; midpoint odometry is as good as the exact arc here; a float heading accumulates a systematic error, and rounding the *step* to an integer accumulates one too, so the integer heading must come from the counts; a tolerance below the sensor's resolution makes a move never finish; calibration fixes the scale of distances and turns and nothing else. Flash and RAM figures are the compiler's for the real chip.

What they do not show. The dead zone, the lag, the weak motor and the radius and track errors are assumptions, and a real robot slips, has backlash, has a floor that changes, and loses encoder counts, none of which is here. No update was timed, so the cost of float in *time* is unmeasured; only its flash and RAM are. The odometry of a real robot drifts from the first metre for reasons this simulator cannot make: the next step, as in the Python course, is to correct it with another sensor (Programming for Robotics with Arduino C++ 10), and before then to drive a real square, mark where it ends, and calibrate on the floor you will use.

CHALLENGE 1: WHICH SPEED ESTIMATOR?

Compare ticks counted over windows of 20, 50 and 100 ms with the period of the last two ticks, at true speeds from 0.01 to 0.5 m/s on the simulator. Print each estimator's rms error as a percentage of the speed, and the delay each adds. Say which you would use below 0.05 m/s, at 0.1 m/s and at 0.5 m/s, and what you give up for it.

View solution

CHALLENGE 2: KEEP THE HEADING HONEST

Find, for steps of 1e-3, 1e-4 and 1e-5 rad, the heading at which a float `th += dth` freezes, and compare it with dth × 2^{24}. Then keep the heading of a wheel pair that gains one tick of difference per update three ways: a float added to and wrapped, a float computed from the count, and a binary angle computed from the count; print the error against a double truth at 10^5 to 10^8 updates, and say which you would choose on the Uno.

View solution

CHALLENGE 3: FIND THE DEAD ZONE

The simulator holds a hidden dead duty that your program may not read. Find it by driving the motor: hold each duty from 0 up for 1.5 s, count the ticks in the last second, and call a duty "moving" at 3 ticks or more. Then command 0.02, 0.05 and 0.1 m/s through the plain mapping and through `speed_to_pwm_dz` with the value you found, for two different hidden dead zones (57 and 23).

View solution

QUICK REFERENCE

- **Dead zone.** With a dead duty of 40 (assumed) the plain mapping gives nothing below 0.1 m/s (0.1 m/s gives 0.026); `speed_to_pwm_dz` is within 2.3 % from 0.05 m/s. One duty step is 2.3 mm/s, the slowest steady speed.
- **Ramp.** Step to 0.5 m/s: peak 5.00 m/s^2, 95 % in 0.299 s. Ramps of 2.0, 1.0, 0.5 m/s^2: peaks 1.84, 0.99, 0.50; times 0.448, 0.637, 1.068 s.
- **Wheel speed.** 0.576 mm per tick. A 20 ms window sees 0 or 28.8 mm/s: at 0.01 m/s it says 0 in 65 of 100 samples (13.74 mm/s rms). The period of two ticks: 0.00 error there, 4.36 mm/s at 0.5 m/s with a 20 µs clock.
- **Odometry.** Floor 0.45 mm and 0.03°, from the ticks. Euler 0.5 to 3.9 mm from 10 to 200 ms; midpoint and exact arc 0.45 to 0.16, equal within 0.02 mm. Use midpoint.
- **Float heading.** Never wrapped: 76° off after 10^5 updates of 0.02 rad. Wrapped: −0.09°, still drifting (−9.07° at 10^7). A float freezes near dth × 2^{24} to twice that (32768 rad for 1e-3). Rounding the step to an integer accumulates too; compute the heading from the count.
- **Integers.** Binary angle (uint16, 65,536 per turn) and micrometres; Q15 quarter-wave table of 65 entries. Same accuracy as float, except a 32-bit product limit: 65 mm per step (185 mm error at 200 ms). Flash: midpoint float +1898, arc +2216, integers +1072; RAM +20, +20, +171 (table not in PROGMEM).
- **Tolerance.** Below the sensor's resolution a move never ends: tolerances 0.003 and 0.006 rad never finish (swing 1.78°); 0.010 rad finishes in 1.46 s.
- **Calibration.** A 2 m drive gave 2.0305 m: radius 0.03350 m (true 0.0335). Four ordered turns gave 3.943: track 0.15216 m (true 0.152). Held straight: 1.0152 m before, 1.0002 m after. The 90° turn: 88.93° before, 88.78° after (stopping error larger than the scale error).
- **Budget.** `robot_ch5.ino`: 8868 bytes flash (27 %), 409 bytes RAM (19 %). Nothing timed.

- **Not run:** anything on a board; the dead zone, lag, weak motor and the radius and track errors are assumptions.

Next: Programming for Robotics with Arduino C++ 6, Control Loops in Code

Programming for Robotics with Arduino C++ 6 (planned; not yet written) replaces the proportional controllers of this chapter with a PID class: anti-windup and a filtered derivative, tuned in the simulator first, a cascaded speed and position loop, a proper heading hold (the steady error of section 6 is what an integral term removes), and a fixed-point PID, with the time per pass counted in microseconds.

Programming for Robotics with Arduino C++

Course 2 · Chapter 6 · Control Loops in Code

Programming for Robotics with Arduino C++ 5 drove the robot with proportional controllers and ended each move with a steady error (a heading hold that left 41 mm of drift) and a turn tolerance chosen by hand. This chapter replaces them with a proper controller: a PID class whose integral does not wind up and whose derivative does not amplify noise, tuned on the simulator and then tried on a robot it was not tuned for, a cascaded position and speed loop, a heading hold that removes the steady error, and an integer version of the controller for a chip with no floating-point unit.

The chapter keeps the habit of the whole course: every claim about a controller is a number from running it, and the numbers are as often *against* the textbook as for it. The cascade, for instance, did not beat a single loop here, and the setting with the best score was the one that broke on a different robot.

WHAT WAS RUN, AND WHAT WAS NOT

Run on a PC (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): every experiment with an output under it and the challenge solutions, on the simulator of Programming for Robotics with Arduino C++ 3: a wheel that follows its command with a 0.1 s lag, an 8-bit duty, and a speed measured from the encoder ticks over 100 ms. The "other robots" (heavier, lighter, a weak motor) are **assumptions I chose**, not measurements. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`): `robot_ch6.ino` and the one-controller-at-a-time sketch. Flash and RAM are the compiler's; the instruction counts come from `avr-objdump`. **Time per pass was not measured**: no board was attached, so the chapter counts instructions and library calls instead of microseconds, and says so where it matters.

1. A PID class, and a rig to test it in

A PID controller turns an error (setpoint minus measurement) into an output as the sum of three terms: **P**, proportional to the error now; **I**, proportional to the error accumulated so far, which removes a steady error; and **D**, proportional to how fast the error is changing, which damps. Written once as a class, it can run a wheel speed, a heading and a distance. In Python the same class is a dozen lines; here the differences are that every number has a type and a cost, and that the sample time is a parameter, not something a thread supplies.

IN PYTHON	ON THE UNO IN C++
`pid = PID(1.5, 6, 0.05)`, state in an object	`Pid pid(config);` a struct of numbers; about 50 bytes of RAM per controller (`sizeof` is 52 on the PC)
`dt = time.time() - last`	`dt` is the fixed control period (0.02 s), passed in: the scheduler of Programming for Robotics with Arduino C++ 1 guarantees it
floats are free	each float add, multiply or divide is a library call; section 7 counts them
an unbounded integer cannot overflow	an integer integral is an `int32_t` with limits chosen on purpose, and with guard bits (section 7)

Here is the whole of it: the float `Pid` with a choice of anti-windup, a filtered derivative that can act on the measurement instead of the error, and two integer versions of a PI(D) that section 7 compares.

```
// pid.h -- control loops (Programming for Robotics with Arduino C++ 6). Plain C++11: no
// Pid is the float controller; PidFixed and PidFixedNaive are integer versions of the s
#pragma once
#include <stdint.h>

inline float pid_clamp(float v, float lo, float hi) { return v < lo ? lo : (v > hi ? hi

enum AntiWindup : uint8_t { AW_NONE = 0, AW_CLAMP = 1, AW_BACKCALC = 2 };

struct PidConfig {
  float kp, ki, kd;               // proportional (output per unit error), integral (pe
  float out_min, out_max;         // the output the actuator can really take
  float d_tau;                    // time constant (s) of the low-pass filter on the de
  bool d_on_measurement;          // true: differentiate the measurement, so a setpoint
  AntiWindup aw;                  // what to do with the integral while the output is s
  float kb;                       // back-calculation gain (1/s), used by AW_BACKCALC
};

class Pid {
 public:
  explicit Pid(const PidConfig& c) : c_(c), i_(0.0f), d_f_(0.0f), prev_e_(0.0f), prev_m_
  void reset() { i_ = 0.0f; d_f_ = 0.0f; first_ = true; }
  float update(float setpoint, float measurement, float dt) {
    float e = setpoint - measurement;
    if (first_) { prev_e_ = e; prev_m_ = measurement; first_ = false; }
    float d_raw = c_.d_on_measurement ? -(measurement - prev_m_) / dt : (e - prev_e_) / 
    if (c_.d_tau > 0.0f) d_f_ += (dt / (c_.d_tau + dt)) * (d_raw - d_f_); else d_f_ = d_
```

```
    prev_e_ = e; prev_m_ = measurement;
    float p = c_.kp * e, d = c_.kd * d_f_;
    float u = p + i_ + d;
    float us = pid_clamp(u, c_.out_min, c_.out_max);
    switch (c_.aw) {
      case AW_NONE:     i_ += c_.ki * e * dt; break;                                  /
      case AW_CLAMP:    if (!((u > c_.out_max && e > 0.0f) || (u < c_.out_min && e < 0.0
      case AW_BACKCALC: i_ += (c_.ki * e + c_.kb * (us - u)) * dt; break;             /
    }
    return us;
  }
  float integral() const { return i_; }
 private:
  PidConfig c_;
  float i_, d_f_, prev_e_, prev_m_;
  bool first_;
};

// ---- the integer PI(D): error and output in the same small units (here mm/s), gains a
// kp_q8 = Kp x 256, ki_dt_q16 = Ki x dt x 65536 (the integral gain already multiplied b
// The integral accumulator keeps 16 fractional bits ("guard bits"): rounding in the con
class PidFixed {
 public:
  PidFixed(int16_t kp_q8, int32_t ki_dt_q16, int16_t kd_q8, int16_t out_min, int16_t out
      : kp_(kp_q8), ki_(ki_dt_q16), kd_(kd_q8), lo_(out_min), hi_(out_max), acc_(0), pre
  int16_t update(int16_t setpoint, int16_t measurement) {
    int32_t e = (int32_t)setpoint - measurement;
    if (first_) { prev_m_ = measurement; first_ = false; }
    int32_t p = (kp_ * e) >> 8;
    int32_t d = -((int32_t)kd_ * (measurement - prev_m_)) >> 8;
    prev_m_ = measurement;
    acc_ += ki_ * e;
    int32_t lo = (int32_t)lo_ * 65536L, hi = (int32_t)hi_ * 65536L;
    if (acc_ > hi) acc_ = hi; else if (acc_ < lo) acc_ = lo;
    int32_t u = p + d + (acc_ >> 16);
    if (u > hi_) u = hi_; else if (u < lo_) u = lo_;
    return (int16_t)u;
  }
  int32_t integral_q16() const { return acc_; }
 private:
  int16_t kp_; int32_t ki_; int16_t kd_; int16_t lo_, hi_;
  int32_t acc_; int16_t prev_m_; bool first_;
};

// THE SAME WITHOUT GUARD BITS: the integral gain times the error is shifted down every
```

```
class PidFixedNaive {
 public:
  PidFixedNaive(int16_t kp_q8, int16_t ki_dt_q8, int16_t out_min, int16_t out_max) : kp_
  int16_t update(int16_t setpoint, int16_t measurement) {
    int32_t e = (int32_t)setpoint - measurement;
    int32_t p = (kp_ * e) >> 8;
    i_ += (ki_ * e) >> 8;
    if (i_ > hi_) i_ = hi_; else if (i_ < lo_) i_ = lo_;
    int32_t u = p + i_;
    if (u > hi_) u = hi_; else if (u < lo_) u = lo_;
    return (int16_t)u;
  }
 private:
  int16_t kp_, ki_, lo_, hi_;
  int32_t i_;
};
```

The experiments all run on one test rig: the simulator, a control step of 20 ms, and a speed estimate from the ticks over the last five control steps (100 ms). That estimate moves in steps of 5.76 mm/s and is on average 50 ms old, so the controller has to live with quantisation and delay, as a real one does.

```
// rig.h -- a test rig for speed and position loops on the simulator (HOST ONLY). One si
// a control step of 20 ms (20 simulator steps of 1 ms), and the measured speed taken fr
// that estimate is 5.76 mm/s and it is on average 50 ms old, which is what a real contr
#pragma once
#include <cstdint>
#include <cmath>
#include "sim.h"
#include "motion.h"

struct SpeedRig {
  SimHW w;
  int32_t hist[6];
  float pt;
  explicit SpeedRig(const RobotParams& rp, uint32_t seed = 1) : w(rp, SensorParams(), Ar
    for (int i = 0; i < 6; ++i) hist[i] = 0;
  }
  // one control step of 20 ms with both wheels commanded `left` and `right` (m/s)
  void step(float left, float right) {
    w.drive(left, right);
    for (int k = 0; k < 20; ++k) w.step(0.001);
```

```
    for (int i = 0; i < 5; ++i) hist[i] = hist[i + 1];
    int32_t l, r; w.read_ticks(l, r);
    hist[5] = l;
  }
  float speed() const { return (float)(hist[5] - hist[0]) * pt / 0.1f; }          // mea
  double true_speed() const { return w.wheel_l(); }
  float distance() const { int32_t l, r; w.read_ticks(l, r); return 0.5f * (float)(l + r
};

inline RobotParams default_robot() { RobotParams rp; rp.tau = 0.1; return rp; }
```

2. Integral windup

The integral term keeps adding while there is an error, and a motor has a limit: it cannot go faster than 0.5 m/s however large the command. If the setpoint is out of reach, the error stays large, the integral keeps growing, and when the setpoint finally comes back into reach the integral is still huge and holds the output at its limit long after the error has gone. That is **windup**. The experiment asks for 0.8 m/s (out of reach) for 1.5 s and then 0.2 m/s, with a PI controller (Kp 1.0, Ki 6.0), once with no protection, once stopping the integral while the output is saturated (clamping), and once with **back-calculation**, which pulls the integral back by the amount the output was cut:

```
// e_windup.cpp -- integral windup on a wheel speed loop (host build). PI controller (Kp
// The setpoint is 0.8 m/s for 1.5 s (more than the motor can give), then 0.2 m/s for 3 s
#include <cstdio>
#include <cmath>
#include "rig.h"
#include "pid.h"

struct Out { double integral_at_drop, recover_s, speed_1s_after, undershoot; };

static Out run(AntiWindup aw, float ki) {
  SpeedRig rig(default_robot());
  PidConfig c = {1.0f, ki, 0.0f, 0.0f, 0.5f, 0.0f, true, aw, 8.0f};
  Pid pid(c);
  Out o = {0, -1, 0, 0};
  const int hold = 75;                                  // 1.5 s of 20 ms steps
  for (int i = 0; i < 75 + 150; ++i) {
    float sp = i < hold ? 0.8f : 0.2f;
    if (i == hold) o.integral_at_drop = pid.integral();
```

```
      float u = pid.update(sp, rig.speed(), 0.02f);
      rig.step(u, u);
      if (i >= hold) {
        double t = (i - hold + 1) * 0.02;
        if (o.recover_s < 0 && rig.true_speed() <= 0.22) o.recover_s = t;
        if (std::fabs(t - 1.0) < 0.011) o.speed_1s_after = rig.true_speed();
        if (rig.true_speed() < 0.2 - o.undershoot) o.undershoot = 0.2 - rig.true_speed();
      }
    }
    return o;
  }

  int main() {
    std::printf("setpoint 0.8 m/s (out of reach: the motor tops out at 0.5) for 1.5 s, ther
    std::printf("  anti-windup            integral when setpoint drops (m/s)   time until
    const char* names[] = {"none", "clamp the integral", "back-calculation"};
    for (int a = 0; a < 3; ++a) {
      Out o = run((AntiWindup)a, 6.0f);
      std::printf("  %-20s   %34.2f   %28.2f   %30.3f   %26.3f\n", names[a], o.integral_at_
    }
    return 0;
  }
```

```
setpoint 0.8 m/s (out of reach: the motor tops out at 0.5) for 1.5 s, then 0.2 m/s; PI, K
  anti-windup            integral when setpoint drops (m/s)   time until speed <= 0.22 (s
  none                                                 3.18                           1.9
  clamp the integral                                   0.23                           0.1
  back-calculation                                     0.42                           0.1
```

With no protection the integral reaches 3.18 (in m/s of output, six times the motor's whole range), and after the setpoint drops to 0.2 the wheel is *still at 0.500 m/s one second later*; it takes 1.98 s to come back within 10 %. Clamping and back-calculation recover in 0.10 and 0.12 s. They are not the same: clamping stops the integral at 0.23 and then undershoots by 0.044 m/s, back-calculation leaves 0.42 in the integral and undershoots 0.003, because it cuts the integral only by what the saturation cut. Challenge 1 sweeps how long the setpoint was out of reach and how hard back-calculation should pull. The practical rule: every controller whose output is limited, which is every controller that drives a motor, needs one of these.

3. The derivative: kick and noise

The derivative term reacts to change, and there are two traps. First, if it differentiates the *error*, a step in the setpoint is a jump in the error and the derivative produces a huge one-step output, the **kick**. Differentiating the *measurement* instead gives the same damping without the kick, because the measurement cannot jump. Second, it amplifies noise: the speed measurement here moves in steps of 5.76 mm/s, and a step in 20 ms looks like a large rate. A low-pass filter on the derivative cuts that. The experiment steps the setpoint from 0.1 to 0.3 m/s at 1 s with Kd = 0.05 s:

```
// e_deriv.cpp -- the derivative term on a noisy, quantised speed measurement (host buil
// The speed measure moves in steps of 5.76 mm/s (one tick over 100 ms) and each step is
// Setpoint: 0.1 m/s for 1 s, then a step to 0.3 m/s. Measured: the command in the first
// (rms of its change from step to step over the last 2 s), and the overshoot of the tru
#include <cstdio>
#include <cmath>
#include "rig.h"
#include "pid.h"

static void run(const char* name, float kd, bool on_meas, float tau) {
  SpeedRig rig(default_robot());
  PidConfig c = {1.5f, 6.0f, kd, 0.0f, 0.5f, tau, on_meas, AW_CLAMP, 8.0f};
  Pid pid(c);
  double first = 0, sq = 0, peak = 0; int n = 0; float prev_u = 0;
  for (int i = 0; i < 300; ++i) {
    float u = pid.update(i < 50 ? 0.1f : 0.3f, rig.speed(), 0.02f);
    if (i == 50) first = u;
    rig.step(u, u);
    if (i >= 200) { double du = u - prev_u; sq += du * du; ++n; }
    prev_u = u;
    if (i >= 50 && rig.true_speed() > peak) peak = rig.true_speed();
  }
  std::printf("  %-34s %12.3f   %20.2f   %15.1f\n", name, first, std::sqrt(sq / n) * 100
}

int main() {
  std::printf("PID on a wheel speed loop, setpoint 0.1 -> 0.3 m/s at 1 s; Kd 0.05 s wher
  std::printf("  derivative                         command at step  step-to-step jitter
  run("none (PI)", 0.0f, true, 0.0f);
  run("on the error, unfiltered", 0.05f, false, 0.0f);
  run("on the measurement, unfiltered", 0.05f, true, 0.0f);
  run("on the measurement, filter 0.05 s", 0.05f, true, 0.05f);
  run("on the measurement, filter 0.10 s", 0.05f, true, 0.10f);
```

```
  return 0;
}
```

```
PID on a wheel speed loop, setpoint 0.1 -> 0.3 m/s at 1 s; Kd 0.05 s where used
  derivative                        command at step  step-to-step jitter (mm/s)   oversh
  none (PI)                                  0.400                      3.89              0
  on the error, unfiltered                   0.500                     14.68              1
  on the measurement, unfiltered             0.378                     17.01              1
  on the measurement, filter 0.05 s          0.402                      6.37              0
  on the measurement, filter 0.10 s          0.402                      5.97              0
```

The kick is plain: the derivative on the error drives the command to 0.500 (the limit) in the first step after the setpoint changes, where the plain PI gives 0.400 and the derivative on the measurement 0.378. The jitter of the settled command (its step-to-step change) is 3.89 mm/s for the PI and 17.01 mm/s with an unfiltered derivative: four times as much, from noise alone. A filter of 0.05 s brings it to 6.37. And look at what the derivative *buys* here: the overshoot of the true speed is 0.6 % without it and 1.4 % with it unfiltered. For a speed loop on a first-order motor the derivative adds noise and gains nothing; the cases where it earns its place (a position loop, section 5) have an extra lag to damp. Challenge 2 sweeps the filter.

4. Tuning in the simulator, and the robot you did not tune for

The simulator is where tuning is cheap, and the experiment below does it the plain way: a grid of Kp and Ki for the speed PI, scored by the IAE (the area between setpoint and true speed; smaller is better) and the overshoot, on a step from 0 to 0.3 m/s. Then the best setting is tried on two other robots, a heavier one with weaker motors and a lighter one with stronger motors (both assumed), as a stand-in for the step from the simulator to the hardware:

```
// e_tune.cpp -- tuning a wheel speed PI in the simulator, then trying it on a robot tha
// Step 0 -> 0.3 m/s, 4 s, integral clamp. Score: IAE, the area between the setpoint and
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "rig.h"
#include "pid.h"

struct Score { double iae, over; };
```

```
static Score run(float kp, float ki, const RobotParams& rp) {
  SpeedRig rig(rp);
  PidConfig c = {kp, ki, 0.0f, 0.0f, 0.5f, 0.0f, true, AW_CLAMP, 8.0f};
  Pid pid(c);
  Score s = {0, 0}; double peak = 0;
  for (int i = 0; i < 200; ++i) {
    float u = pid.update(0.3f, rig.speed(), 0.02f);
    rig.step(u, u);
    s.iae += std::fabs(0.3 - rig.true_speed()) * 0.02;
    if (rig.true_speed() > peak) peak = rig.true_speed();
  }
  s.over = (peak - 0.3) / 0.3 * 100.0;
  return s;
}

int main() {
  RobotParams nominal = default_robot();
  std::printf("grid on the nominal robot (lag 0.1 s): IAE (m/s x s) / overshoot (%%)\n")
  std::printf("      Kp / Ki        2            5           10           20\n");
  double best_iae = 1e9, best_kp = 0, best_ki = 0, any_iae = 1e9, any_kp = 0, any_ki = 0
  for (float kp : {0.5f, 1.0f, 2.0f, 4.0f, 8.0f}) {
    std::printf("  %8.1f", kp);
    for (float ki : {2.0f, 5.0f, 10.0f, 20.0f}) {
      Score s = run(kp, ki, nominal);
      std::printf("   %6.3f/%5.1f", s.iae, s.over);
      if (s.iae < any_iae) { any_iae = s.iae; any_kp = kp; any_ki = ki; }
      if (s.over <= 10.0 && s.iae < best_iae) { best_iae = s.iae; best_kp = kp; best_ki =
    }
    std::printf("\n");
  }
  std::printf("lowest IAE of all: Kp %.1f, Ki %.0f (%.3f); lowest IAE with overshoot of

  RobotParams heavy = default_robot(); heavy.tau = 0.2; heavy.gain_l = heavy.gain_r = 0.
  RobotParams light = default_robot(); light.tau = 0.05; light.gain_l = light.gain_r = 1
  std::printf("\nthe lowest-IAE setting and a gentler one on three robots, IAE / oversho
  std::printf("  setting                nominal          heavy, weak motors   light, s
  for (int k = 0; k < 2; ++k) {
    float kp = k == 0 ? (float)any_kp : 1.0f, ki = k == 0 ? (float)any_ki : 5.0f;
    Score a = run(kp, ki, nominal), b = run(kp, ki, heavy), c = run(kp, ki, light);
    std::printf("  Kp %.1f, Ki %2.0f          %6.3f / %5.1f    %6.3f / %5.1f       %6.3
  }
  (void)best_kp; (void)best_ki; (void)best_iae;
```

```
  return 0;
}
```

```
grid on the nominal robot (lag 0.1 s): IAE (m/s x s) / overshoot (%)
      Kp / Ki        2            5           10           20
       0.5    0.129/ -0.0    0.040/  0.4    0.047/ 28.7    0.109/ 37.2
       1.0    0.127/ -0.4    0.040/  0.5    0.022/ 13.3    0.043/ 31.8
       2.0    0.137/ -1.8    0.057/  0.6    0.029/  0.6    0.020/  7.5
       4.0    0.127/ 14.5    0.071/ 14.5    0.053/ 14.5    0.062/ 14.9
       8.0    0.253/ 23.9    0.245/ 25.5    0.241/ 27.3    0.235/ 26.9
lowest IAE of all: Kp 2.0, Ki 20 (0.020); lowest IAE with overshoot of at most 10 %: Kp
```

```
  setting                nominal           heavy, weak motors   light, strong motors
  Kp 2.0, Ki 20          0.020 /   7.5     0.036 /   5.5         0.091 /  45.6
  Kp 1.0, Ki  5          0.040 /   0.5     0.055 /   0.2         0.030 /   1.7
```

The lowest IAE in the grid is Kp 2, Ki 20 (0.020, with 7.5 % overshoot), and it is the best of 20 settings. On the heavy robot it still works (0.036, 5.5 %). On the light robot, whose motors are stronger and quicker, it overshoots by 45.6 % and its IAE is 4.5 times worse: the loop gain went up by 25 % and the lag went down, and a setting tuned to the edge of stability on the first robot is over the edge on the second. The gentler Kp 1, Ki 5 scores twice the IAE on the nominal robot (0.040) and holds on all three (overshoots of 0.5, 0.2 and 1.7 %). The lesson is about what to optimise: a controller that will be moved to hardware should be tuned for *margin* (an overshoot well under the limit, a setting in the middle of a flat region of the grid), then checked on the perturbed plants, not for the best score on the one model.

The grid also shows a feature of this loop that a tidy theory would not predict: at Kp 4 the overshoot is 14.5 % whatever Ki is, and at Kp 8 it is 24–27 %. That is the 50 ms delay of the speed estimate: past some Kp the loop is fighting a measurement that is half a tick old. The simulator knows about the delay because the rig puts it there; a simulator that measured the speed directly would not have shown this.

5. Cascaded loops: do they help?

A distance move can be one loop, a PID on the distance that commands the wheels directly, or a **cascade**: an outer loop on the distance whose output is the setpoint of an inner speed loop,

which is the PI of section 4. The textbook arguments for the cascade are that each loop is simple and fast, the inner loop hides the motor from the outer one, and a limit on the speed is one clamp. The experiment drives 1 m both ways, single loop with a filtered derivative and cascade with a proportional outer loop, and scores the true distance:

```
// e_cascade.cpp -- driving 1 m: one position loop that commands the motor directly, aga
// Single loop: PID on the distance (Kd 0.3 s on the measurement, filtered 0.05 s), outp
// Cascade: outer proportional loop on the distance gives a speed setpoint (limited to +
// Scored on the TRUE distance (the simulator's own position): overshoot, and the time a
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "rig.h"
#include "pid.h"

struct R { double over_mm, settle_s, peak_v; };

static R score(SpeedRig& rig, double peak_v, double over, double last_outside) { (void)r

static R single(float kp, const RobotParams& rp = default_robot()) {
  SpeedRig rig(rp);
  PidConfig c = {kp, 0.0f, 0.3f, -0.4f, 0.4f, 0.05f, true, AW_CLAMP, 8.0f};
  Pid pid(c);
  double over = 0, peak_v = 0, last_out = 0;
  for (int i = 0; i < 400; ++i) {
    float u = pid.update(1.0f, rig.distance(), 0.02f);
    rig.step(u, u);
    double x = rig.w.x - 0.3;
    if ((x - 1.0) * 1000 > over) over = (x - 1.0) * 1000;
    if (rig.true_speed() > peak_v) peak_v = rig.true_speed();
    if (std::fabs(x - 1.0) > 0.005) last_out = (i + 1) * 0.02;
  }
  return score(rig, peak_v, over, last_out);
}

static R cascade(float kp_pos, const RobotParams& rp = default_robot()) {
  SpeedRig rig(rp);
  PidConfig inner_c = {1.0f, 5.0f, 0.0f, -0.4f, 0.4f, 0.0f, true, AW_CLAMP, 8.0f};
  Pid inner(inner_c);
  double over = 0, peak_v = 0, last_out = 0;
  for (int i = 0; i < 400; ++i) {
    float v_sp = pid_clamp(kp_pos * (1.0f - rig.distance()), -0.4f, 0.4f);
    float u = inner.update(v_sp, rig.speed(), 0.02f);
    rig.step(u, u);
```

```
    double x = rig.w.x - 0.3;
    if ((x - 1.0) * 1000 > over) over = (x - 1.0) * 1000;
    if (rig.true_speed() > peak_v) peak_v = rig.true_speed();
    if (std::fabs(x - 1.0) > 0.005) last_out = (i + 1) * 0.02;
  }
  return score(rig, peak_v, over, last_out);
}

int main() {
  std::printf("a 1 m move: overshoot of the true distance (mm), time to stay within 5 mm
  std::printf("  loop                              overshoot mm   settled at (s)   peak
  for (float kp : {2.0f, 4.0f, 8.0f}) { R r = single(kp); std::printf("  single position
  for (float kp : {1.0f, 2.0f, 4.0f}) { R r = cascade(kp); std::printf("  cascade, outer
  RobotParams heavy = default_robot(); heavy.tau = 0.2; heavy.gain_l = heavy.gain_r = 0.
  RobotParams light = default_robot(); light.tau = 0.05; light.gain_l = light.gain_r = 1
  std::puts(""); std::printf("the same two settings (single Kp 4.0; cascade outer Kp 2.0
  std::printf("  robot                     single loop        cascade"); std::puts("");
  const char* names[] = {"nominal", "heavy, weak motors", "light, strong motors"};
  RobotParams robots[] = {default_robot(), heavy, light};
  for (int k = 0; k < 3; ++k) {
    R a = single(4.0f, robots[k]), b = cascade(2.0f, robots[k]);
    std::printf("  %-22s   %7.1f / %5.2f    %7.1f / %5.2f", names[k], a.over_mm, a.settl
  }
  return 0;
}
```

```
a 1 m move: overshoot of the true distance (mm), time to stay within 5 mm (s; 8.00 = nev
  loop                              overshoot mm   settled at (s)   peak speed
  single position PID, Kp 2.0                0.0            4.18        0.40
  single position PID, Kp 4.0                0.0            3.04        0.40
  single position PID, Kp 8.0                2.3            2.64        0.40
  cascade, outer Kp 1.0                      0.0            5.16        0.40
  cascade, outer Kp 2.0                      2.6            3.22        0.40
  cascade, outer Kp 4.0                      9.4            3.50        0.40

the same two settings (single Kp 4.0; cascade outer Kp 2.0) on other robots: overshoot m
  robot                     single loop        cascade
  nominal                      0.0 /  3.04        2.6 /  3.22
  heavy, weak motors           3.3 /  3.52        2.3 /  3.72
  light, strong motors         0.0 /  2.58        1.2 /  3.32
```

On this plant **the cascade did not help**. The single loop with Kp 4 reaches 1 m with no overshoot in 3.04 s; the cascade's best settings overshoot 2.6 mm and settle at 3.22 s, and with the outer Kp at 4 it overshoots 9.4 mm. On the other robots the two are within a few millimetres of each other, and the single loop settles up to 0.7 s sooner. What the table does support is the safe-speed argument: both loops are clamped at 0.4 m/s and both reached it. What it does not test is the argument that the inner loop absorbs a change in the motor (a flat battery, a heavier load) while the outer loop is untouched; the simulator runs with fixed parameters, so a mid-move disturbance was not tried. I would not choose a cascade because a book does; I would choose it when the speed loop is needed anyway, as it is for the wheel speeds of Programming for Robotics with Arduino C++ 5 and the heading hold of section 6.

6. Heading hold

Programming for Robotics with Arduino C++ 5 left a P-only heading hold with 41 mm of drift and −2.6°: a proportional controller against a steady disturbance (the weak right motor) always leaves a steady error. The integral term removes it. The experiment drives straight for 4 s (1.6 m) at 0.4 m/s with the right motor 5 % weak, on the odometry heading, and scores the true end state:

```
// e_heading.cpp -- holding a heading while driving straight at 0.4 m/s for 4 s (1.6 m),
// The heading comes from the encoders: (right ticks - left ticks) x per_tick / track, e
// the output is a turn rate (limited to +-1.5 rad/s) that is split across the wheels. S
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "rig.h"
#include "pid.h"

static void run(const char* name, float kp, float ki, float kd) {
  RobotParams rp = default_robot(); rp.gain_r = 0.95;
  SpeedRig rig(rp);
  PidConfig c = {kp, ki, kd, -1.5f, 1.5f, 0.05f, true, AW_CLAMP, 8.0f};
  Pid pid(c);
  const float track = (float)rp.track;
  double worst = 0;
  for (int i = 0; i < 200; ++i) {
    int32_t l, r; rig.w.read_ticks(l, r);
    float th = (float)(r - l) * rig.pt / track;
    float w = pid.update(0.0f, th, 0.02f);
```

```
    const float v = 0.4f;
    rig.step(v - 0.5f * w * track, v + 0.5f * w * track);
    double lat = (rig.w.y - 1.0) * 1000;
    if (std::fabs(lat) > worst) worst = std::fabs(lat);
  }
  std::printf("  %-26s   %17.2f   %17.1f   %17.1f\n", name, rig.w.th * 180 / SIM_PI, (ri
}

int main() {
  std::printf("straight 1.6 m with the right motor 5 %% weak; true values at the end\n")
  std::printf("  heading controller           heading (deg)     sideways (mm)     worst
  run("none (open loop)", 0.0f, 0.0f, 0.0f);
  run("P only, Kp 3", 3.0f, 0.0f, 0.0f);
  run("PI, Kp 3, Ki 1", 3.0f, 1.0f, 0.0f);
  run("PI, Kp 3, Ki 3", 3.0f, 3.0f, 0.0f);
  run("PI, Kp 3, Ki 8", 3.0f, 8.0f, 0.0f);
  run("PID, Kp 3, Ki 3, Kd 0.2", 3.0f, 3.0f, 0.2f);
  return 0;
}
```

```
straight 1.6 m with the right motor 5 % weak; true values at the end
  heading controller           heading (deg)     sideways (mm)     worst sideways (mm)
  none (open loop)                     -29.80            -386.8                   386.8
  P only, Kp 3                          -2.70             -66.5                    66.5
  PI, Kp 3, Ki 1                        -0.76             -39.6                    39.6
  PI, Kp 3, Ki 3                        -0.03             -17.8                    17.8
  PI, Kp 3, Ki 8                         0.10              -6.4                     8.2
  PID, Kp 3, Ki 3, Kd 0.2               -0.03             -18.0                    18.0
```

Open loop the robot ends 387 mm to the side at −29.8°. P only leaves −2.70° and 66.5 mm (similar to the P-only hold of Programming for Robotics with Arduino C++ 5, which left −2.6° and 41 mm on a 1 m move). Adding Ki = 1 gives −0.76°, Ki = 3 gives −0.03° and 17.8 mm, Ki = 8 gives 0.10° and 8.2 mm worst excursion. The integral removes the heading error but not the sideways drift already built up while it was catching up, which is why the position error falls more slowly than the heading error. The derivative (Kd = 0.2) changes nothing visible here, as in section 3.

7. A PID with integers

On a chip with no floating-point unit a float PI is expensive, and `PidFixed` in the header is the integer alternative: error and output in the same small units (mm/s), the gains as scaled integers (Kp×256, Ki×dt×65536), and an integral accumulator that keeps 16 fractional bits. The accumulator clamp is the anti-windup. The variant `PidFixedNaive` does what looks natural, shifting the product down each step, and it has the same bug as the integer exponential filter of Programming for Robotics with Arduino C++ 4: **a small error adds nothing, and the integral stalls**. The experiment runs the three on the speed loop for 20 s:

```
// e_fixed.cpp -- the same speed PI in float, in integers with guard bits, and in integer
// Scored by the TRUE speed: mean error over the last 2 s of a 20 s run, and (against th
#include <cstdio>
#include <cmath>
#include <vector>
#include <initializer_list>
#include "rig.h"
#include "pid.h"

struct R { double err_mm_s; std::vector<float> cmd; };

enum Kind { FLOAT, FIXED, NAIVE };

static R run(Kind kind, float ki) {
  SpeedRig rig(default_robot());
  PidConfig c = {1.5f, ki, 0.0f, 0.0f, 0.5f, 0.0f, true, AW_CLAMP, 8.0f};
  Pid pf(c);
  PidFixed px((int16_t)std::lround(1.5 * 256), (int32_t)std::lround(ki * 0.02 * 65536), 
  PidFixedNaive pn((int16_t)std::lround(1.5 * 256), (int16_t)std::lround(ki * 0.02 * 256
  R r; r.err_mm_s = 0; int n = 0;
  for (int i = 0; i < 1000; ++i) {
    float m = rig.speed();
    float u;
    if (kind == FLOAT) u = pf.update(0.3f, m, 0.02f);
    else {
      int16_t meas = (int16_t)std::lround(m * 1000.0f);
      int16_t out = kind == FIXED ? px.update(300, meas) : pn.update(300, meas);
      u = out / 1000.0f;
    }
    r.cmd.push_back(u);
    rig.step(u, u);
    if (i >= 900) { r.err_mm_s += (0.3 - rig.true_speed()) * 1000.0; ++n; }
  }
  r.err_mm_s /= n;
  return r;
```

```
}

int main() {
  std::printf("speed PI, 20 s; steady error = mean of (300 - true speed) in mm/s over th
  std::printf("  Ki (1/s)   float: error   guard bits: error   diff   no guard bits: err
  for (float ki : {5.0f, 1.0f, 0.3f}) {
    R f = run(FLOAT, ki), x = run(FIXED, ki), nv = run(NAIVE, ki);
    double dx = 0, dn = 0;
    for (size_t i = 0; i < f.cmd.size(); ++i) { dx = std::fmax(dx, std::fabs(f.cmd[i] - 
    std::printf("  %8.1f   %12.2f   %17.2f   %4.1f   %20.2f   %4.1f\n", ki, f.err_mm_s, 
  }
  return 0;
}
```

```
speed PI, 20 s; steady error = mean of (300 - true speed) in mm/s over the last 2 s; dif
  Ki (1/s)   float: error   guard bits: error   diff   no guard bits: error   diff
       5.0          -0.02               -0.44   30.0                  5.29   30.0
       1.0           0.13               -0.42   17.5                 44.57   60.5
       0.3          11.91               12.00   18.2                116.49   117.2
```

With Ki = 5 the float and guard-bit controllers end within 0.5 mm/s of the setpoint; the controller without guard bits ends 5.29 mm/s off. With Ki = 1 the float controller ends 0.13 mm/s off, the guard-bit one −0.42, and the controller without guard bits **44.57 mm/s off**: the product of Ki×dt (a count of 5 in Q8) and a small error shifts down to zero, and the integral stops. With Ki = 0.3 it is 116.49 mm/s off, against 11.91 for the float (which is only slow to converge; its time constant is long). The two integer controllers differ from the float one by up to 30 mm/s in the command, but that is a transient: the measurement is rounded to whole millimetres per second and the rounding changes the path a little; the end values agree. Challenge 3 finds how many fractional bits the integral gain needs: with 4 bits Ki = 1 rounds to a count of zero and the integral does nothing at all (a steady error of 120.55 mm/s, exactly the P-only value), and 12 bits get the gain to within 2 %.

What does the integer controller save? The two versions were compiled alone into a sketch that reads A0 and prints the command (the first row is the sketch without a controller), and the machine code of the update function was read with `avr-objdump`:

```
arduino-cli compile --fqbn arduino:avr:uno --warnings all -DVARIANT_x s_pid
(whole sketch; "base" only reads A0 and prints it; the update function is noinline and i
```

```
  variant                          flash bytes   RAM bytes   adds flash   adds RAM   ins
  base (no controller)                    1890         188            0          0
  PI in float (Pid)                       4130         265         2240         77
  PI in integers (PidFixed)               2496         207          606         19

(the instruction counts are of all the code in the function, not of one pass through it;
```

The float PI costs 2240 bytes of flash and 77 of RAM and its update function is 350 instructions with 27 calls into the float library; the integer one costs 606 and 19, with 167 instructions and 3 calls (all integer multiplications). That is the bill in space and in work; the bill in *time* is not in the table because no time was measured. Each of those 27 calls runs a subroutine of many instructions, but how many microseconds a pass takes, and whether it matters at a 20 ms period, is a measurement that needs the board. Counting calls tells you where to look first, not how long it takes.

8. The whole controller, and the budget

`robot_ch6.ino` puts it together: an outer proportional loop on the distance gives a speed setpoint, a heading PI gives a turn rate that is split across the wheels, and each wheel has its own speed PI. The gains are the ones the simulator chose (the speed PI of section 4, the heading PI of section 6, an outer Kp of 2).

```
// robot_ch6.ino -- Programming for Robotics with Arduino C++, chapter 6: a 1 m move witl
// Compile-checked for the Uno (arduino:avr:uno). NOT run on a board: no board, motor or
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include "scheduler.h"
#include "conv.h"
#include "pins.h"
#include "motion.h"
#include "pid.h"

const float V_MAX_MPS = 0.5f;
const uint8_t DEAD_DUTY = 40;                          // ASSUMED: measure yours (Prograr
const float WHEEL_R = 0.033f, TRACK = 0.150f;          // replace by the calibrated valu
const float PER_TICK = 6.2831853f * WHEEL_R / 360.0f;  // 360 counts per turn (2x decodi

volatile int32_t g_ticks_l = 0, g_ticks_r = 0;
```

```
ISR(INT0_vect) { bool a = (PIND & _BV(PD2)) != 0, b = (PINC & _BV(PC1)) != 0; g_ticks_l -
ISR(INT1_vect) { bool a = (PIND & _BV(PD3)) != 0, b = (PINC & _BV(PC2)) != 0; g_ticks_r -

static void read_ticks(int32_t& l, int32_t& r) { ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { l =
static void drive(float vl, float vr) {
  digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
  digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
  analogWrite(PIN_ML_PWM, speed_to_pwm_dz(vl >= 0.0f ? vl : -vl, V_MAX_MPS, DEAD_DUTY));
  analogWrite(PIN_MR_PWM, speed_to_pwm_dz(vr >= 0.0f ? vr : -vr, V_MAX_MPS, DEAD_DUTY));
}

// one speed loop per wheel: speed from the ticks over the last five control steps (100 m
struct WheelLoop {
  Pid pid;
  int32_t hist[6];
  WheelLoop() : pid({1.0f, 5.0f, 0.0f, -0.4f, 0.4f, 0.0f, true, AW_CLAMP, 8.0f}) { for (u
  float update(float sp, int32_t ticks) {
    for (uint8_t i = 0; i < 5; ++i) hist[i] = hist[i + 1];
    hist[5] = ticks;
    float speed = (float)(hist[5] - hist[0]) * PER_TICK / 0.1f;
    return pid.update(sp, speed, 0.02f);
  }
};
static WheelLoop left_loop, right_loop;
static Pid heading_pid({3.0f, 3.0f, 0.0f, -1.5f, 1.5f, 0.05f, true, AW_CLAMP, 8.0f});   /
static Scheduler<3> sch;
static bool moving = true;
static float g_dist = 0.0f, g_head = 0.0f;

static void control_task(void*, uint32_t) {
  int32_t l, r; read_ticks(l, r);
  g_dist = 0.5f * (float)(l + r) * PER_TICK;
  g_head = (float)(r - l) * PER_TICK / TRACK;
  if (!moving) { drive(0.0f, 0.0f); return; }
  float v_sp = pid_clamp(2.0f * (1.0f - g_dist), -0.4f, 0.4f);           // outer loop: a
  float w = heading_pid.update(0.0f, g_head, 0.02f);                      // heading hol
  float vl = left_loop.update(v_sp - 0.5f * w * TRACK, l);
  float vr = right_loop.update(v_sp + 0.5f * w * TRACK, r);
  drive(vl, vr);
  if (g_dist > 0.995f && g_dist < 1.005f && fabsf(v_sp) < 0.02f) moving = false;
}
static void report_task(void*, uint32_t) {
  Serial.print(F("dist ")); Serial.print(g_dist, 3); Serial.print(F(" m  heading ")); Se
}
```

```
void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT); pinMode(PIN_ML_PWM, OUTPUT);
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP); pinMode(PIN_ENC_L_
  EICRA = _BV(ISC00) | _BV(ISC10); EIFR = _BV(INTF0) | _BV(INTF1); EIMSK = _BV(INT0) | _
  sch.add(control_task, 0, 20);
  sch.add(report_task, 0, 500);
}

void loop() { sch.run(millis()); }
```

It compiles for the Uno with no warnings under `--warnings all` to **7638 bytes of flash (23 %) and 541 bytes of RAM (26 %)**: smaller than Programming for Robotics with Arduino C++ 5's sketch in flash (it has no `Mover` and no `sinf`) and 132 bytes larger in RAM (three `Pid` objects and two histories). It was *not run*. The same controller, with the same gains and steps, was run on the simulator:

```
// e_compose.cpp -- the whole controller of robot_ch6.ino on the simulator: position P (
// Same gains, same 20 ms step, same 100 ms speed estimate as the sketch. A 1 m move, ru
#include <cstdio>
#include <cmath>
#include "rig.h"
#include "pid.h"

static void run(const char* name, const RobotParams& rp) {
  SpeedRig rig(rp);
  PidConfig sc = {1.0f, 5.0f, 0.0f, -0.4f, 0.4f, 0.0f, true, AW_CLAMP, 8.0f};
  Pid left(sc), right(sc);
  PidConfig hc = {3.0f, 3.0f, 0.0f, -1.5f, 1.5f, 0.05f, true, AW_CLAMP, 8.0f};
  Pid heading(hc);
  int32_t hl[6] = {0}, hr[6] = {0};
  const float track = 0.150f;                          // what the code believes; the s
  double done_at = -1;
  bool moving = true;
  for (int i = 0; i < 400; ++i) {
    int32_t l, r; rig.w.read_ticks(l, r);
    float dist = 0.5f * (float)(l + r) * rig.pt, head = (float)(r - l) * rig.pt / track;
    for (int k = 0; k < 5; ++k) { hl[k] = hl[k + 1]; hr[k] = hr[k + 1]; }
    hl[5] = l; hr[5] = r;
    float sl = (float)(hl[5] - hl[0]) * rig.pt / 0.1f, sr = (float)(hr[5] - hr[0]) * rig
    if (!moving) { rig.step(0, 0); continue; }
    float v_sp = pid_clamp(2.0f * (1.0f - dist), -0.4f, 0.4f);
```

```
      float w = heading.update(0.0f, head, 0.02f);
      float vl = left.update(v_sp - 0.5f * w * track, sl, 0.02f);
      float vr = right.update(v_sp + 0.5f * w * track, sr, 0.02f);
      rig.step(vl, vr);
      if (dist > 0.995f && dist < 1.005f && std::fabs(v_sp) < 0.02f) { moving = false; don
    }
    std::printf("  %-26s   %12.4f   %12.1f   %12.2f   %10.2f\n", name, rig.w.x - 0.3, (rig
  }

  int main() {
    RobotParams weak = default_robot(); weak.gain_r = 0.95;
    RobotParams heavy = default_robot(); heavy.tau = 0.2; heavy.gain_l = heavy.gain_r = 0.
    std::printf("a 1 m move with the controller of robot_ch6.ino; true end state\n");
    std::printf("  robot                        distance (m)   sideways (mm)   heading (de
    run("nominal", default_robot());
    run("right motor 5 % weak", weak);
    run("heavy, motors 20 % weaker", heavy);
    return 0;
  }
```

```
a 1 m move with the controller of robot_ch6.ino; true end state
  robot                        distance (m)   sideways (mm)   heading (deg)   done at (s
  nominal                            0.9992           0.0           0.00         3.26
  right motor 5 % weak               0.9989         -23.2           3.28         3.34
  heavy, motors 20 % weaker          1.0035           0.0           0.00         3.76
```

On the nominal robot the 1 m move ends 0.9992 m out, straight, in 3.26 s. With the right motor 5 % weak it ends 0.9989 m out but 23.2 mm to the side with a heading of 3.28°, which is not what the heading hold of section 6 promised. Printing the trace explains it: during the cruise the heading drifts to −2.9° and the integral builds up to correct it; the move then slows toward the end, the integral that was right at 0.4 m/s is too much at 0.05 m/s, the heading swings through zero to +3.7°, and the move ends (the stop condition looks at the distance and the speed setpoint, not the heading) before it settles. The integral time constant, Kp/Ki = 1 s, is long for a move of 3 s. Either a longer straight, a heading integral that is reset or frozen when the speed setpoint falls, or a stop condition that also waits for the heading, would help; I did not try them. This is the kind of thing a simulator is for: it found a failure of a plan that looked finished.

9. What the numbers can and cannot be trusted for

What they show. The shapes: an unprotected integral winds up (3.18 in the integral, 1.98 s to recover) and clamping or back-calculation fixes it; a derivative of the error kicks and an unfiltered one multiplies the noise by four and bought nothing on a first-order plant; the best-scoring tuning broke on a lighter robot; the cascade was no better than a single loop; the integral removes a steady heading error but needs time; an integer integral without guard bits stalls. The flash, RAM and instruction counts are the compiler's for the real chip.

What they do not show. The plants (lag, gains, delay) and the "other robots" are assumptions: real motors have a dead zone, a stiction threshold, a battery that sags and a gearbox with backlash, none of which the loop here has met. The cascade was tested on fixed parameters, so its disturbance-rejection argument is untested. **No time per pass was measured**, so the cost of float is known in space and in library calls and not in microseconds. Every gain in `robot_ch6.ino` came from the simulator, and a real robot needs them re-tuned by the procedure of section 4: start gentle, raise Kp until it just starts to ring, back off, add Ki, and test on a perturbed setup before trusting it.

CHALLENGE 1: THE PRICE OF WINDUP

Sweep how long the setpoint stays out of reach (0.25, 0.5, 1 and 2 s) and measure how long the true speed takes to return to within 10 % of 0.2 m/s, for no anti-windup, clamping, and back-calculation with gains of 1, 4, 8 and 20 per second. Explain why the unprotected recovery time grows with the time out of reach, and what happens to back-calculation when kb is too small.

View solution

CHALLENGE 2: FILTERING THE DERIVATIVE

Sweep the time constant of the derivative's filter from 0 to 0.5 s on the speed PID of section 3 (derivative on the measurement) and print the command jitter, the overshoot and the 90 % rise time. Choose a value and say what you give up for it, and why the rise-time column is not a clean trend.

View solution

CHALLENGE 3: BITS FOR THE INTEGRAL

Write an integer PI whose integral gain has F fractional bits. For Ki of 1.0 and 0.3 per second and F of 4, 6, 8, 10, 12 and 16, print the gain it actually implements (as a percentage of the wanted one) and the

steady speed error after 30 s. Find the smallest F for each Ki, and say what goes wrong at the bottom of the table.

View solution

QUICK REFERENCE

- **Windup.** Unprotected integral 3.18 after 1.5 s out of reach; 1.98 s to recover, still 0.500 m/s a second later. Clamp 0.10 s (undershoot 0.044), back-calculation 0.12 s (0.003). Every limited output needs one.
- **Derivative.** On the error it kicks the command to the limit (0.500); on the measurement 0.378. Unfiltered it gives 17.01 mm/s of jitter against 3.89 for the PI; a 0.05 s filter 6.37. On a first-order speed loop it bought nothing (overshoot 0.6 % without, 1.4 % with).
- **Tuning.** Best IAE on the model: Kp 2, Ki 20 (0.020, 7.5 %); on a lighter robot 45.6 % overshoot, IAE 0.091. Gentler Kp 1, Ki 5: 0.040 and under 2 % on all three. Tune for margin, then test on perturbed plants. Kp past about 4 hits the 50 ms measurement delay.
- **Cascade.** Single loop Kp 4: 0.0 mm overshoot, 3.04 s. Cascade outer Kp 2: 2.6 mm, 3.22 s. No benefit found; the disturbance argument was not tested.
- **Heading hold.** 1.6 m with a 5 % weak motor: open loop −29.8°, 387 mm; P −2.70°, 66.5 mm; PI Ki 3 −0.03°, 17.8 mm. The integral needs time: in the 1 m move of `robot_ch6.ino` it overshoots to +3.28° at the stop.
- **Integer PI.** Guard bits (Q16 accumulator): error −0.42 mm/s at Ki 1; without them 44.57. At 4 fractional bits Ki 1 rounds to a count of 0 and the integral is dead (120.55 mm/s). 12 bits: within 2 % of the gain.
- **Cost.** Float PI +2240 bytes flash, +77 RAM, 350 instructions, 27 float-library calls; integer PI +606, +19, 167 instructions, 3 integer multiplies. Time per pass: not measured.
- **Budget.** `robot_ch6.ino`: 7638 bytes flash (23 %), 541 bytes RAM (26 %).
- **Not run:** anything on a board; every plant parameter and every gain comes from the simulator.

Next: Programming for Robotics with Arduino C++ 7, State Machines and Behaviours

Programming for Robotics with Arduino C++ 7 (planned; not yet written) gives the robot something to decide: `enum class` states and `switch`, table-driven machines with function pointers, timers and time-outs, and line-follower, obstacle-avoider and wall-follower behaviours built on the moves and controllers of this chapter, tested on the host before they are tried on the board, with the transition table held in flash or static memory instead of a heap.

Programming for Robotics with Arduino C++

Course 2 · Chapter 7 · State Machines and Behaviours

Programming for Robotics with Arduino C++ 6 gave the robot controllers: given a goal (a speed, a heading, a distance) they drive the error to zero. This chapter gives it *decisions*. Something is in the way, so back up and turn; the wall has ended, so search for it; the line has gone, so look for it for a while and then give up. Decisions like these are a **state machine**: a small set of states, the events that move the robot from one state to another, and what the robot does in each state.

The chapter builds one machine three ways (a `switch`, a table in RAM, a table in flash), compares what they cost on the Uno, writes the tests a machine needs before it goes near a robot (does the table behave like the switch? is there a state it cannot leave?), and then builds a wall follower and a line follower on the same pattern. As in the whole course, every number is from running the code, and where the numbers embarrass the design the chapter says so: the wall follower's side error turns out to be an oscillation of the controller itself, and one setting of the avoider's dropout rule was counting in the wrong unit.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): every experiment with an output under it and the three challenge solutions, on the simulator of Programming for Robotics with Arduino C++ 3, extended for this chapter with a second range sensor looking to the right, a circular line on the floor and a five-sensor line array. The arena, the line, the sensor noise, the dropouts and the sensor array's gains and offsets are **assumptions I chose**, not measurements. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`): the size-test sketch in its four variants and `robot_ch7.ino`. The flash and RAM figures are the compiler's for the real chip. **Time per pass was not measured**, and nothing was attached to a board: no sensor, motor or bumper. The warnings printed by the Arduino core's own `new.cpp` (unused parameter `tag`) are the core's, not the sketch's.

1. A machine with `enum class` and `switch`

The obstacle avoider has three states. **CRUISE**: drive forward; if the bumper is pressed or something is closer than 0.25 m, go to BACKUP. **BACKUP**: reverse for 400 ms, then pick a random direction and a random turn time between 400 and 1200 ms and go to TURN. **TURN**: spin until the time is up, then go back to CRUISE. In Python you would write the state as an

`Enum` member or a string and branch with `if` / `match`; in C++ the state is an `enum class` (a distinct type, so a state cannot be mixed up with an event or a number) stored in one byte, and the branch is a `switch`, whose cases the compiler can check: with `-Wall` it warns about a state you forgot to handle.

IN PYTHON	ON THE UNO IN C++
`class State(Enum): CRUISE = 1 ...`	`enum class AvoidState : uint8_t { CRUISE, BACKUP, TURN };` one byte, a type of its own
`match state:` or an `if` chain	`switch (st_)`; a missing case is a compiler warning
a dict of state to function, built when the program starts, on the heap	an array of function pointers, a `const` table fixed at compile time, which can live in flash instead of RAM (section 3)
`time.time()` for "how long have I been here"	the time is passed in (`now_ms`) and the state remembers when it was entered (`t0_`); the machine never reads a clock, so a PC can test it
an exception when something goes wrong	an explicit FAULT state and a time-out that leads to it (section 5)

Two design choices run through every behaviour in this chapter. First, a behaviour is a **pure function of what the sensors said and the time**: `update(Sense, now)` returns wheel speeds (`Cmd`), and it never touches a pin, so it can be driven by the simulator, by a test, or by the real sensors with no change. Second, the tick on which the state *changes* returns zero speeds, so a transition never carries out the old state's action and the new one's in the same step.

Here is the whole header: the avoider as a `switch`, the same machine as a table (section 3), the wall follower (section 4) and the line follower (section 6). Its one decision about sensor trouble is `NearCheck`: what to do when the front sensor returns "no echo". The *naive* rule reads no echo as "nothing there"; the *safe* rule holds the last good reading for `drop_limit` control ticks and then assumes something *is* there. **The unit matters and it is the control tick (20 ms), not the sensor reading (60 ms):** `drop_limit` 3 holds for one reading.

```
// fsm.h -- state machines and behaviours (Programming for Robotics with Arduino C++ 7).
// Behaviours are pure: they take what the sensors said (Sense) and the time, and return
#pragma once
#include <stdint.h>
#include "calib.h"

#if defined(__AVR__) && !defined(FSM_NO_PROGMEM)            // define FSM_NO_PROGMEM to
#include <avr/pgmspace.h>
```

```
#define FSM_PROGMEM PROGMEM
inline uint8_t fsm_read8(const uint8_t* p) { return pgm_read_byte(p); }
#define FSM_READ_PTR(p) pgm_read_ptr(p)
#else
#define FSM_PROGMEM
inline uint8_t fsm_read8(const uint8_t* p) { return *p; }
#define FSM_READ_PTR(p) (*(p))
#endif

struct Sense { float front_m; float right_m; bool bump; };   // distances in metres; nega
struct Cmd { float vl, vr; };

// a tiny random generator, so a behaviour that picks a random turn is reproducible from
struct Xor32 {
  uint32_t s;
  explicit Xor32(uint32_t seed) : s(seed ? seed : 1u) {}
  uint32_t next() { s ^= s << 13; s ^= s >> 17; s ^= s << 5; return s; }
};

inline Cmd spin(float w, float track) { Cmd c = {-0.5f * w * track, 0.5f * w * track}; re
inline Cmd arc(float v, float w, float track) { Cmd c = {v - 0.5f * w * track, v + 0.5f *

// ---- "is something close?": what to do with a front reading of -1 (no echo) ----
// naive: no echo means nothing there (far). safe: hold the last good reading for drop_li
class NearCheck {
 public:
  NearCheck(float stop_m, uint8_t drop_limit, bool safe) : stop_(stop_m), limit_(drop_lim
  bool near(float front_m, bool bump) {
    if (front_m < 0.0f) {
      if (drops_ < 250) ++drops_;
      if (!safe_) return bump;                          // naive: only the bumper can say
      if (drops_ > limit_) return true;                 // too long without an echo: assum
    } else { drops_ = 0; last_ = front_m; }
    return bump || last_ < stop_;
  }
 private:
  float stop_; uint8_t limit_; bool safe_; float last_; uint8_t drops_;
};

// ============ 1. a switch-based machine: an obstacle avoider ============
enum class AvoidState : uint8_t { CRUISE, BACKUP, TURN };

struct AvoidConfig {
  float v, stop_m, back_v, turn_w, track;
  uint16_t back_ms, turn_min_ms, turn_max_ms;
```

```
  uint8_t drop_limit;
  bool safe;
};

class Avoider {
 public:
  Avoider(const AvoidConfig& c, uint32_t seed) : c_(c), near_(c.stop_m, c.drop_limit, c.
  Cmd update(const Sense& s, uint32_t now_ms) {
    bool near = near_.near(s.front_m, s.bump);
    uint32_t el = now_ms - t0_;
    switch (st_) {
      case AvoidState::CRUISE:
        if (near) { st_ = AvoidState::BACKUP; t0_ = now_ms; break; }
        return arc(c_.v, 0.0f, c_.track);
      case AvoidState::BACKUP:
        if (el >= c_.back_ms) {
          st_ = AvoidState::TURN; t0_ = now_ms;
          uint32_t r = rng_.next();
          dur_ = c_.turn_min_ms + (r >> 8) % (uint32_t)(c_.turn_max_ms - c_.turn_min_ms +
          dir_ = (r & 1) ? 1.0f : -1.0f;
          break;
        }
        return arc(-c_.back_v, 0.0f, c_.track);
      case AvoidState::TURN:
        if (el >= dur_) { st_ = AvoidState::CRUISE; t0_ = now_ms; break; }
        return spin(dir_ * c_.turn_w, c_.track);
    }
    Cmd stop = {0.0f, 0.0f};                          // the tick on which the state ch
    return stop;
  }
  AvoidState state() const { return st_; }
 private:
  AvoidConfig c_;
  NearCheck near_;
  Xor32 rng_;
  AvoidState st_;
  uint32_t t0_, dur_;
  float dir_;
};

// ============ 2. the same machine as a TABLE: rows of (state, event, next state, actio
// Events are bits of a mask; for the current state the first row (in table order) whose
namespace tbl {
enum State : uint8_t { CRUISE, BACKUP, TURN, NSTATES };
enum Event : uint8_t { EV_BUMP, EV_NEAR, EV_TIMER };
```

```
enum Action : uint8_t { A_NONE, A_START_BACK, A_START_TURN };

const uint8_t ROWS[][4] FSM_PROGMEM = {                // from,        event,    to,
  { CRUISE, EV_BUMP,  BACKUP, A_START_BACK },
  { CRUISE, EV_NEAR,  BACKUP, A_START_BACK },
  { BACKUP, EV_TIMER, TURN,   A_START_TURN },
  { TURN,   EV_TIMER, CRUISE, A_NONE       },
};
const uint8_t NROWS = sizeof(ROWS) / sizeof(ROWS[0]);

// the engine: returns the action of the row that fired (A_NONE if none did, and `fired`
inline uint8_t step(const uint8_t (*rows)[4], uint8_t nrows, uint8_t& state, uint8_t eve
  for (uint8_t i = 0; i < nrows; ++i) {
    if (fsm_read8(&rows[i][0]) == state && ((events >> fsm_read8(&rows[i][1])) & 1)) {
      state = fsm_read8(&rows[i][2]);
      fired = true;
      return fsm_read8(&rows[i][3]);
    }
  }
  fired = false;
  return A_NONE;
}
}  // namespace tbl

struct AvoidCtx;
typedef Cmd (*StateFn)(const AvoidCtx&);
struct AvoidCtx { AvoidConfig c; float dir; };
inline Cmd do_cruise(const AvoidCtx& x) { return arc(x.c.v, 0.0f, x.c.track); }
inline Cmd do_backup(const AvoidCtx& x) { return arc(-x.c.back_v, 0.0f, x.c.track); }
inline Cmd do_turn(const AvoidCtx& x) { return spin(x.dir * x.c.turn_w, x.c.track); }
const StateFn DO_FN[tbl::NSTATES] FSM_PROGMEM = { do_cruise, do_backup, do_turn };

class TableAvoider {
 public:
  TableAvoider(const AvoidConfig& c, uint32_t seed, const uint8_t (*rows)[4] = tbl::ROWS
      : near_(c.stop_m, c.drop_limit, c.safe), rng_(seed), state_(tbl::CRUISE), t0_(0), 
  Cmd update(const Sense& s, uint32_t now_ms) {
    uint8_t ev = 0;
    bool near = near_.near(s.front_m, s.bump);
    if (s.bump) ev |= 1 << tbl::EV_BUMP;
    if (near) ev |= 1 << tbl::EV_NEAR;
    uint32_t el = now_ms - t0_;
    uint32_t limit = state_ == tbl::BACKUP ? x_.c.back_ms : dur_;
    if (state_ != tbl::CRUISE && el >= limit) ev |= 1 << tbl::EV_TIMER;
    bool fired;
```

```
    uint8_t act = tbl::step(rows_, nrows_, state_, ev, fired);
    if (fired) {
      t0_ = now_ms;
      if (act == tbl::A_START_TURN) {
        uint32_t r = rng_.next();
        dur_ = x_.c.turn_min_ms + (r >> 8) % (uint32_t)(x_.c.turn_max_ms - x_.c.turn_min_
        x_.dir = (r & 1) ? 1.0f : -1.0f;
      }
      Cmd stop = {0.0f, 0.0f};
      return stop;
    }
    StateFn f = (StateFn)FSM_READ_PTR(&DO_FN[state_]);
    return f(x_);
  }
  uint8_t state() const { return state_; }
 private:
  NearCheck near_;
  Xor32 rng_;
  uint8_t state_;
  uint32_t t0_, dur_;
  const uint8_t (*rows_)[4];
  uint8_t nrows_;
  AvoidCtx x_;
};

// ============ 3. a wall follower (the wall on the RIGHT) ============
enum class WallState : uint8_t { FOLLOW, CORNER, SEARCH, FAULT };
struct WallConfig {
  float v, target_m, kp, corner_front_m, corner_clear_m, corner_w, search_v, search_w, t
  uint16_t side_hold_ms, corner_timeout_ms, search_timeout_ms;
  bool full;                                          // false: the naive version (ste
};

class WallFollower {
 public:
  explicit WallFollower(const WallConfig& c) : c_(c), st_(WallState::FOLLOW), t0_(0), si
  Cmd update(const Sense& s, uint32_t now) {
    if (s.right_m >= 0.0f) { side_ = s.right_m; side_t_ = now; }
    bool side_ok = (now - side_t_) <= c_.side_hold_ms;               // a reading no old
    bool front_near = s.bump || (s.front_m >= 0.0f && s.front_m < c_.corner_front_m);
    if (!c_.full) return arc(c_.v, side_ok ? -c_.kp * (side_ - c_.target_m) : 0.0f, c_.t
    uint32_t el = now - t0_;
    switch (st_) {
      case WallState::FOLLOW:
        if (front_near) { st_ = WallState::CORNER; t0_ = now; break; }
```

```
        if (!side_ok) { st_ = WallState::SEARCH; t0_ = now; break; }
        return arc(c_.v, -c_.kp * (side_ - c_.target_m), c_.track);
      case WallState::CORNER:
        if (s.front_m > c_.corner_clear_m) { st_ = WallState::FOLLOW; t0_ = now; break;
        if (el > c_.corner_timeout_ms) { st_ = WallState::FAULT; break; }
        return spin(c_.corner_w, c_.track);
      case WallState::SEARCH:
        if (front_near) { st_ = WallState::CORNER; t0_ = now; break; }
        if (side_ok) { st_ = WallState::FOLLOW; t0_ = now; break; }
        if (el > c_.search_timeout_ms) { st_ = WallState::FAULT; break; }
        return arc(c_.search_v, -c_.search_w, c_.track);
      case WallState::FAULT:
        break;
    }
    Cmd stop = {0.0f, 0.0f};
    return stop;
  }
  WallState state() const { return st_; }
 private:
  WallConfig c_;
  WallState st_;
  uint32_t t0_;
  float side_;
  uint32_t side_t_;
};

// ============ 4. a line follower ============
enum class LineState : uint8_t { FOLLOW, SEARCH, FAULT };
struct LineConfig { float v, k_w, slow, search_w, track; uint16_t search_timeout_ms; };

class LineFollower {
 public:
  explicit LineFollower(const LineConfig& c) : c_(c), st_(LineState::FOLLOW), t0_(0), la
  LineCal<5>& cal() { return cal_; }
  Cmd update(const int16_t* raw, uint32_t now) {
    static const int16_t POS[5] = {40, 20, 0, -20, -40};      // mm, left positive
    for (uint8_t i = 0; i < 5; ++i) scaled_[i] = cal_.scaled(i, raw[i]);
    float pos = 0.0f;
    bool seen = line_position<5>(scaled_, POS, pos);
    Cmd stop = {0.0f, 0.0f};
    switch (st_) {
      case LineState::FOLLOW:
        if (!seen) { st_ = LineState::SEARCH; t0_ = now; ++lost_events_; break; }
        last_side_ = pos >= 0 ? 1.0f : -1.0f;
        { float a = pos < 0 ? -pos : pos; float v = c_.v * (1.0f - c_.slow * (a > 40.0f
```

```
          return arc(v, c_.k_w * pos, c_.track); }
      case LineState::SEARCH:
        if (seen) { st_ = LineState::FOLLOW; break; }
        if (now - t0_ > c_.search_timeout_ms) { st_ = LineState::FAULT; break; }
        return spin(last_side_ * c_.search_w, c_.track);
      case LineState::FAULT:
        break;
    }
    return stop;
  }
  LineState state() const { return st_; }
  uint32_t lost_events() const { return lost_events_; }
 private:
  LineConfig c_;
  LineState st_;
  uint32_t t0_;
  float last_side_;
  uint32_t lost_events_;
  LineCal<5> cal_;
  int16_t scaled_[5];
};
```

2. Trying it: how an obstacle avoider behaves

The experiment runs the avoider for 60 s in the 3 × 2 m arena, 200 seeded runs per row, and scores the bumper presses (collisions), the share of the arena's 600 cells of 10 cm that the robot visited, and the seconds spent outside the walls (a robot that has pushed through is a failed robot). The sensor is read every 60 ms; the control tick is 20 ms.

```
// e_avoid.cpp -- a wandering obstacle avoider in the 3 x 2 m arena, 60 s, 200 seeded ru
// Machine: CRUISE -> (something near or bumper) -> BACKUP for 400 ms -> TURN for 400..1
// "naive": a reading of -1 (no echo) means nothing is there. "safe": hold the last good
// Scored: collisions (the bumper going from released to pressed), and the share of the
#include <cstdio>
#include <cmath>
#include <set>
#include <initializer_list>
#include "sim.h"
#include "fsm.h"

struct R { double bumps, coverage, stuck; };
```

```
static R run(bool safe, double dropout, int seeds, int limit = 3) {
  R r = {0, 0, 0};
  for (int seed = 1; seed <= seeds; ++seed) {
    RobotParams rp; rp.tau = 0.1;
    SensorParams sp; sp.dropout = dropout;
    SimHW w(rp, sp, Arena(), seed);
    w.th = (seed % 8) * 0.785;
    AvoidConfig c = {0.3f, 0.25f, 0.2f, 2.0f, 0.15f, 400, 400, 1200, (uint8_t)limit, saf
    Avoider av(c, (uint32_t)seed);
    Sense s = {-1.0f, -1.0f, false};
    bool was_bumped = false;
    std::set<int> cells;
    double bumps = 0, outside = 0;
    for (int tick = 0; tick < 3000; ++tick) {
      uint32_t now = tick * 20;
      if (tick % 3 == 0) { s.front_m = w.range_m(); }
      s.bump = w.bumped();
      Cmd cmd = av.update(s, now);
      w.drive(cmd.vl, cmd.vr);
      for (int k = 0; k < 20; ++k) w.step(0.001);
      bool b = w.bumped();
      if (b && !was_bumped) ++bumps;
      was_bumped = b;
      if (w.x > 0 && w.x < 3.0 && w.y > 0 && w.y < 2.0) cells.insert((int)(w.x / 0.1) * 
    }
    r.bumps += bumps; r.coverage += cells.size() / 600.0; r.stuck += outside;
  }
  r.bumps /= seeds; r.coverage = 100.0 * r.coverage / seeds; r.stuck /= seeds;
  return r;
}

int main() {
  std::printf("obstacle avoider, 60 s, mean over 200 seeded runs\n");
  std::printf("  front sensor dropouts   handling   collisions per run   arena visited (%
  for (double dp : {0.02, 0.30}) for (bool safe : {false, true}) {
    R r = run(safe, dp, 200);
    std::printf("  %20.0f %%   %-8s   %18.2f   %17.1f   %25.2f\n", dp * 100, safe ? "saf
  }
  for (int lim : {6, 10}) { R r = run(true, 0.30, 200, lim); std::printf("  %20.0f %%   
  return 0;
}
```

```
obstacle avoider, 60 s, mean over 200 seeded runs
  front sensor dropouts   handling   collisions per run   arena visited (%)   seconds ou
                    2 %   naive                    0.08                22.4
                    2 %   safe                     0.03                22.3
                   30 %   naive                    0.10                22.4
                   30 %   safe                     0.02                 8.1
                   30 %   safe, hold 6             0.00                16.5
                   30 %   safe, hold 10            0.03                20.6
```

Collisions are rare in every row, between 0 and 0.10 per run, because the only obstacles are the arena walls and the sensor sees the wall almost always; at 30 % dropouts the naive rule does not collide more (0.10) than at 2 % (0.08), and I would not read a difference in those hundredths. What does change is *coverage*: safe handling with a hold of 3 ticks (one reading) visits 8.1 % of the arena at 30 % dropouts against 22.4 % for the naive rule, because the robot keeps seeing "something near" where nothing is and spends its time backing and turning. Raising the hold to 6 ticks recovers 16.5 %, 10 ticks 20.6 %. So the safe rule buys fewer collisions at the price of coverage, and the hold is the knob; challenge 2 sweeps it. The safe, hold-3 row also spent 0.16 s per run outside the arena, which no other row did: on some runs the robot got through a wall. I did not trace why, so it stays a loose end, and a reminder that the number to watch is not only the average.

3. The same machine as a table

Three states fit a `switch` well. At fifteen they do not: the transitions are scattered through the cases, and adding one means editing code. A **table-driven** machine holds the transitions as *data*: one row per transition, `{from, event, to, action}`, and one small engine that looks up the row. In `fsm.h` the rows are four bytes each, the events are bits of a mask (bumper, something near, timer expired), and the first row whose state and event match wins:

```
const uint8_t ROWS[][4] FSM_PROGMEM = {                 // from,        event,    to,
  { CRUISE, EV_BUMP,  BACKUP, A_START_BACK },
  { CRUISE, EV_NEAR,  BACKUP, A_START_BACK },
  { BACKUP, EV_TIMER, TURN,   A_START_TURN },
  { TURN,   EV_TIMER, CRUISE, A_NONE       },
};
```

What the robot does *in* a state is a function, and the table of functions (`DO_FN`, one pointer per state) is also `const`. On the Uno, `PROGMEM` keeps those tables in flash, and they are read with `pgm_read_byte` and `pgm_read_ptr` instead of a plain load, because the Uno's flash and RAM are separate address spaces. That is the point of the "no dynamic allocation" comparison with Python: the whole machine is a few dozen bytes of constants known at compile time, with no heap and no object to build when the program starts.

What does it cost? The same avoider was compiled for the Uno in four ways into a sketch that reads A3 and the bumper pin and writes two PWM outputs (so the compiler cannot discard the machine). The first row has no machine at all:

VARIANT	FLASH (BYTES)	RAM (BYTES)	OVER THE BASELINE
no machine (baseline)	2158	9	
`switch` (`Avoider`)	3402	93	+1244 flash, +84 RAM
table, tables in RAM	3850	118	+1692 flash, +109 RAM
table, tables in flash (`PROGMEM`)	3876	96	+1718 flash, +87 RAM

Two things stand out. The table version is **474 bytes of flash larger than the switch** for this three-state machine: the engine, the lookup and the function-pointer call are code that a hand-written `switch` does not need, and it only pays when the machine has many more states or rows. And putting the tables in flash saves exactly **22 bytes of RAM** (118 against 96): the 16 bytes of the four rows plus the 6 bytes of the three function pointers, at a cost of 26 bytes more flash for the `pgm_read` calls. Twenty-two bytes is a minor saving for this table (about 1 % of the Uno's 2048), but a table of 60 rows of four bytes is 240 bytes, and that is where the flash tables earn their place. The figures include the config struct and the state of `NearCheck`, which every machine variant carries.

```
// s_fsm.ino -- size test for chapter 7: the same obstacle avoider as a switch, as a tabl
// Compile-checked for the Uno only; NOT run on a board. The "sensor" is analogRead(A3) a
// VARIANT 0: no machine (the baseline), 1: switch, 2: table. Add -DFSM_NO_PROGMEM to ke
#include "fsm.h"
#include "pins.h"

#ifndef VARIANT
#define VARIANT 1
#endif
```

```
static AvoidConfig CFG = {0.3f, 0.25f, 0.2f, 2.0f, 0.15f, 400, 400, 1200, 3, true};
#if VARIANT == 1
static Avoider machine(CFG, 1);
#elif VARIANT == 2
static TableAvoider machine(CFG, 1);
#endif

void setup() {
  pinMode(PIN_ML_PWM, OUTPUT); pinMode(PIN_MR_PWM, OUTPUT); pinMode(PIN_BUMP, INPUT_PULL
}

void loop() {
  Sense s;
  s.front_m = (float)analogRead(A3) * (4.0f / 1023.0f);
  s.right_m = -1.0f;
  s.bump = digitalRead(PIN_BUMP) == LOW;
  uint32_t now = millis();
#if VARIANT == 0
  Cmd c = {s.front_m, s.bump ? 0.0f : s.front_m};
  (void)now;
#else
  Cmd c = machine.update(s, now);
#endif
  analogWrite(PIN_ML_PWM, (int)((c.vl < 0 ? -c.vl : c.vl) * 100.0f));
  analogWrite(PIN_MR_PWM, (int)((c.vr < 0 ? -c.vr : c.vr) * 100.0f));
  delay(20);
}
```

4. Testing a machine on the host

A state machine has two kinds of bug that a robot is a bad place to find: one in what it does (a wrong speed), and one in its *structure* (a state with no way out). Because the behaviours never touch a pin, both can be tested on a PC in a second. The experiment below does three things.

Part 1: does the table behave like the switch? Two simulators with the same seed, one driving the switch avoider and one the table avoider, are compared command by command (the bytes of the two `Cmd`s) over 50 seeds at each of three settings. **Part 2: a checker for the table itself**: rows that point outside the table, a repeated (state, event) pair (the second can never fire), states not reachable from CRUISE, and states from which CRUISE can never be reached again. It is run

on the good table and on three tables with one mistake each, and each is then driven in the simulator to see what the mistake looks like on a robot. **Part 3: a time-out**, in section 5.

```
// e_fsmcheck.cpp -- testing state machines on the host (no robot, no hardware).
// Part 1: the switch-based Avoider and the table-driven TableAvoider must give the SAME
// Part 2: a checker that reads a transition table and looks for: rows that point outsid
//         from CRUISE, and states from which CRUISE can never be reached again. It is r
//         and each table is then also driven in the simulator to show what the mistake
// Part 3: a timeout. The line follower is put 200 mm away from the line (it cannot find
//         that fits its 16-bit field (65.5 s) it spins for the whole run.
#include <cstdio>
#include <cstring>
#include <cmath>
#include <set>
#include "sim.h"
#include "fsm.h"

using namespace tbl;

// ---------- part 2: the table checker ----------
struct Report { int out_of_range, duplicates, unreachable, cannot_return; };

static Report check(const uint8_t (*rows)[4], int n, int nstates, int start, bool verbos
  Report r = {0, 0, 0, 0};
  for (int i = 0; i < n; ++i) {
    if (rows[i][0] >= nstates || rows[i][2] >= nstates) { ++r.out_of_range; if (verbose)
    for (int j = 0; j < i; ++j)
      if (rows[i][0] == rows[j][0] && rows[i][1] == rows[j][1]) { ++r.duplicates; if (ve
  }
  bool reach[8] = {false};
  reach[start] = true;
  for (int pass = 0; pass < nstates; ++pass)
    for (int i = 0; i < n; ++i) if (rows[i][0] < nstates && rows[i][2] < nstates && reac
  bool back[8] = {false};                                                    // which st
  back[start] = true;
  for (int pass = 0; pass < nstates; ++pass)
    for (int i = 0; i < n; ++i) if (rows[i][0] < nstates && rows[i][2] < nstates && back
  for (int s = 0; s < nstates; ++s) {
    if (!reach[s]) { ++r.unreachable; if (verbose) std::printf("    state %d cannot be r
    if (!back[s]) { ++r.cannot_return; if (verbose) std::printf("    state %d can never
  }
  return r;
}
```

```
const uint8_t BAD_MISSING[][4] = {                      // the TURN -> CRUISE row was f
  { CRUISE, EV_BUMP,  BACKUP, A_START_BACK },
  { CRUISE, EV_NEAR,  BACKUP, A_START_BACK },
  { BACKUP, EV_TIMER, TURN,   A_START_TURN },
};
const uint8_t BAD_TYPO[][4] = {                         // BACKUP's timer row goes to BA
  { CRUISE, EV_BUMP,  BACKUP, A_START_BACK },
  { CRUISE, EV_NEAR,  BACKUP, A_START_BACK },
  { BACKUP, EV_TIMER, BACKUP, A_START_TURN },
  { TURN,   EV_TIMER, CRUISE, A_NONE       },
};
const uint8_t BAD_DUP[][4] = {                          // a second CRUISE/EV_NEAR row,
  { CRUISE, EV_BUMP,  BACKUP, A_START_BACK },
  { CRUISE, EV_NEAR,  BACKUP, A_START_BACK },
  { CRUISE, EV_NEAR,  TURN,   A_START_TURN },
  { BACKUP, EV_TIMER, TURN,   A_START_TURN },
  { TURN,   EV_TIMER, CRUISE, A_NONE       },
};

static AvoidConfig cfg(bool safe) { AvoidConfig c = {0.3f, 0.25f, 0.2f, 2.0f, 0.15f, 400

struct Sim { double coverage, cruise_pct, bumps; int final_state; };
static Sim drive_table(const uint8_t (*rows)[4], int n, int seeds) {
  Sim out = {0, 0, 0, 0};
  for (int seed = 1; seed <= seeds; ++seed) {
    RobotParams rp; rp.tau = 0.1;
    SimHW w(rp, SensorParams(), Arena(), seed);
    w.th = (seed % 8) * 0.785;
    TableAvoider av(cfg(true), (uint32_t)seed, rows, (uint8_t)n);
    Sense s = {-1.0f, -1.0f, false};
    std::set<int> cells; int cruise = 0; bool was = false;
    for (int tick = 0; tick < 3000; ++tick) {
      if (tick % 3 == 0) s.front_m = w.range_m();
      s.bump = w.bumped();
      Cmd c = av.update(s, tick * 20);
      w.drive(c.vl, c.vr);
      for (int k = 0; k < 20; ++k) w.step(0.001);
      bool b = w.bumped(); if (b && !was) out.bumps += 1; was = b;
      if (av.state() == CRUISE) ++cruise;
      if (w.x > 0 && w.x < 3.0 && w.y > 0 && w.y < 2.0) cells.insert((int)(w.x / 0.1) * 
    }
    out.coverage += cells.size() / 600.0; out.cruise_pct += cruise / 3000.0; out.final_s
  }
  out.coverage = 100.0 * out.coverage / seeds; out.cruise_pct = 100.0 * out.cruise_pct /
  return out;
```

```
}

// ---------- part 1: equivalence ----------
static void equivalence(double dropout, bool safe) {
  int diverged = 0; long ticks = 0; long first = -1;
  for (int seed = 1; seed <= 50; ++seed) {
    SensorParams sp; sp.dropout = dropout; RobotParams rp; rp.tau = 0.1;
    SimHW w1(rp, sp, Arena(), seed), w2(rp, sp, Arena(), seed);
    w1.th = w2.th = (seed % 8) * 0.785;
    Avoider a(cfg(safe), (uint32_t)seed); TableAvoider b(cfg(safe), (uint32_t)seed);
    Sense s1 = {-1.0f, -1.0f, false}, s2 = s1;
    bool bad = false;
    for (int tick = 0; tick < 3000 && !bad; ++tick) {
      if (tick % 3 == 0) { s1.front_m = w1.range_m(); s2.front_m = w2.range_m(); }
      s1.bump = w1.bumped(); s2.bump = w2.bumped();
      Cmd c1 = a.update(s1, tick * 20), c2 = b.update(s2, tick * 20);
      if (std::memcmp(&c1, &c2, sizeof c1) != 0) { bad = true; if (first < 0) first = ti
      w1.drive(c1.vl, c1.vr); w2.drive(c2.vl, c2.vr);
      for (int k = 0; k < 20; ++k) { w1.step(0.001); w2.step(0.001); }
      ++ticks;
    }
    if (bad) ++diverged;
  }
  std::printf("  %8.0f %%   %-5s   %12d   %16ld   %s\n", dropout * 100, safe ? "safe" :
  if (diverged) std::printf("    %d runs diverged, first at tick %ld\n", diverged, first
}

// ---------- part 3: a timeout ----------
static void timeout_demo(uint16_t timeout_ms) {
  RobotParams rp; rp.tau = 0.1;
  SimHW w(rp, SensorParams(), Arena(), 7);
  w.line.on = true;
  LineConfig c = {0.25f, 0.05f, 0.5f, 1.5f, 0.15f, timeout_ms};
  LineFollower f(c);
  w.x = w.line.cx + w.line.R; w.y = w.line.cy; w.th = SIM_PI / 2;
  int16_t raw[5];
  for (int i = -70; i <= 70; ++i) { double x0 = w.x; w.x = x0 + i * 0.001; w.line_read(r
  w.x = w.line.cx + w.line.R + 0.2; w.y = w.line.cy; w.th = SIM_PI / 2;       // 200 mm
  std::printf("  search timeout %5u ms:", (unsigned)timeout_ms);
  const double marks[] = {1, 2, 4, 10, 30};
  int mi = 0; double turned = 0, prev = w.th;
  for (int tick = 0; tick < 1500; ++tick) {
    w.line_read(raw);
    Cmd cmd = f.update(raw, tick * 20);
    w.drive(cmd.vl, cmd.vr);
```

```
    for (int k = 0; k < 20; ++k) w.step(0.001);
    turned += std::fabs(w.th - prev); prev = w.th;
    if (mi < 5 && (tick + 1) * 0.02 >= marks[mi] - 1e-9) {
      std::printf("  %2.0f s: %-6s", marks[mi], f.state() == LineState::FOLLOW ? "FOLLOW
      ++mi;
    }
  }
  std::printf("   turned %.1f revolutions in total\n", turned / (2 * SIM_PI));
}

int main() {
  std::printf("part 1: does the table behave like the switch? (60 s per run, 3000 contro
  std::printf("  dropouts   mode    runs compared   ticks compared   result\n");
  equivalence(0.02, false); equivalence(0.02, true); equivalence(0.30, true);

  std::printf("\npart 2: the table checker\n");
  struct { const char* name; const uint8_t (*rows)[4]; int n; } T[] = {
    {"good table", ROWS, (int)NROWS},
    {"TURN -> CRUISE row missing", BAD_MISSING, (int)(sizeof BAD_MISSING / 4)},
    {"BACKUP timer row typed as BACKUP", BAD_TYPO, (int)(sizeof BAD_TYPO / 4)},
    {"duplicate CRUISE/EV_NEAR row", BAD_DUP, (int)(sizeof BAD_DUP / 4)},
  };
  std::printf("  states: 0 CRUISE, 1 BACKUP, 2 TURN; events: 0 BUMP, 1 NEAR, 2 TIMER\n")
  for (auto& t : T) {
    Report r = check(t.rows, t.n, NSTATES, CRUISE, false);
    std::printf("  %s: %d rows, problems found: %d\n", t.name, t.n, r.out_of_range + r.du
    check(t.rows, t.n, NSTATES, CRUISE, true);
  }
  std::printf("\n  the same tables driven in the simulator (60 s, 50 seeds, 2%% dropouts
  std::printf("  table                                arena visited (%%)   time in CRUISE
  for (auto& t : T) { Sim s = drive_table(t.rows, t.n, 50); std::printf("  %-34s   %16.1

  std::printf("\npart 3: line follower 200 mm off the line (it spins in place looking fo
  timeout_demo(3000);
  timeout_demo(65535);
  return 0;
}
```

```
part 1: does the table behave like the switch? (60 s per run, 3000 control ticks)
  dropouts   mode    runs compared   ticks compared   result
         2 %   naive            50            150000   identical in every run
         2 %   safe             50            150000   identical in every run
        30 %   safe             50            150000   identical in every run
```

```
part 2: the table checker
  states: 0 CRUISE, 1 BACKUP, 2 TURN; events: 0 BUMP, 1 NEAR, 2 TIMER
  good table: 4 rows, problems found: 0
  TURN -> CRUISE row missing: 3 rows, problems found: 2
    state 1 can never get back to state 0 (a trap)
    state 2 can never get back to state 0 (a trap)
  BACKUP timer row typed as BACKUP: 4 rows, problems found: 2
    state 1 can never get back to state 0 (a trap)
    state 2 cannot be reached from state 0
  duplicate CRUISE/EV_NEAR row: 5 rows, problems found: 1
    row 2 repeats (state 0, event 1) of row 1: row 2 can never fire

  the same tables driven in the simulator (60 s, 50 seeds, 2% dropouts, safe handling)
  table                              arena visited (%)   time in CRUISE (%)   collision
  good table                                      22.9                 72.0
  TURN -> CRUISE row missing                       1.4                  4.3
  BACKUP timer row typed as BACKUP                 3.2                  4.3
  duplicate CRUISE/EV_NEAR row                    22.9                 72.0

part 3: line follower 200 mm off the line (it spins in place looking for it)
  search timeout  3000 ms:   1 s: SEARCH   2 s: SEARCH   4 s: FAULT   10 s: FAULT   30 s
  search timeout 65535 ms:   1 s: SEARCH   2 s: SEARCH   4 s: SEARCH  10 s: SEARCH  30 s
```

Part 1: the two implementations produced bit-identical commands in all 150 000 control ticks at each of three settings (2 % and 30 % dropouts, naive and safe). That proves the two agree on those inputs; it does not prove they agree on inputs the simulator did not produce, which is what the checker is for.

Part 2: the checker reports nothing for the good table and one or two problems for each broken one. The three bugs look very different on the robot. With the **TURN → CRUISE row missing**, the robot reaches its first obstacle, backs up, turns, and then spins in TURN forever: 1.4 % of the arena visited and 4.3 % of the time in CRUISE (the time before the first obstacle), against 22.9 % and 72.0 % for the good table. With the **BACKUP row typed as BACKUP**, it reverses forever (by the code, blindly), visits 3.2 % and ends every run with a collision (1.00 per run, against 0.02). The **duplicate row**, a copy-and-paste of the CRUISE / near row with a different target, is the instructive one: the simulator shows *no difference at all* (22.9 %, 72.0 %, 0.02, the same digits as the good table) because the engine takes the first matching row and the copy is never reached. A test that only watches behaviour cannot see it; the checker flags it in a line. That is the argument for checking a table as *data*: some bugs are not behaviours until the day someone reorders the rows.

5. Time-outs: every state needs a way out

A machine that waits for something (an echo, a wall, a line) must not wait forever. The rule that follows from the checker is: **every state that depends on the world must also have an exit that depends only on the clock**, and the last resort is a FAULT state that stops the robot and says so. Part 3 of the experiment puts the line follower 200 mm away from the line, where spinning in place cannot find it (the sensors sweep a circle of 80 mm radius around the robot), and compares a 3 s search time-out with the longest one the 16-bit field allows, 65.5 s:

With the 3 s time-out the robot is in SEARCH at 1 s and 2 s, in FAULT from 4 s on, and has turned 0.7 revolutions in total (3 s at the search rate of 1.5 rad/s). With 65 535 ms it is still in SEARCH at 30 s and has turned 7.1 revolutions: a robot that spins on the spot until the battery is flat, which for a robot that is not on a line is not harmless. The FAULT state is not an admission of failure but a design decision: a robot that stops is safe, and the person who reads “FAULT” on the serial port knows where to look. Note also what a time-out costs: a state with a time-out has to remember when it was entered (`t0_`), and the rows of a table machine need a TIMER event, which section 3’s engine gets from comparing `now - t0_` to the state’s limit.

6. Following a wall

The wall follower keeps the wall on its right. Its states: FOLLOW (steer on the side distance), CORNER (the front is blocked: spin left until the front is clear), SEARCH (the wall has gone: arc to the right to find it), FAULT (a time-out in CORNER or SEARCH). The side sensor is read every 60 ms and its last good reading is held for 300 ms, so a single missing echo does not send the machine into SEARCH. The experiment compares the *naive* controller (steer on the side distance and nothing else) with the machine, at 2 % and at 30 % dropouts, over 100 seeded runs of 100 s each:

```
// e_wall.cpp -- a right-hand wall follower around the 3 x 2 m arena, 100 s at 0.3 m/s,
// "naive": steer on the right-hand distance only (P control, target 0.30 m); nothing el
// FAULT (stop after a time-out). Both sensors are read every 60 ms; the side reading is
// Scored: laps of the arena, collisions (bumper pressed), the rms error of the true sid
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "fsm.h"
```

```
struct R { double laps, bumps, rms_mm, faults; };

static R run(bool full, double dropout, int seeds) {
  R r = {0, 0, 0, 0};
  double sq = 0; long n = 0;
  for (int seed = 1; seed <= seeds; ++seed) {
    RobotParams rp; rp.tau = 0.1;
    SensorParams sp; sp.dropout = dropout;
    SimHW w(rp, sp, Arena(), seed);
    w.x = 0.5; w.y = 0.3; w.th = 0.0;
    WallConfig c = {0.3f, 0.30f, 4.0f, 0.40f, 0.70f, 1.5f, 0.15f, 0.8f, 0.15f, 300, 3000
    WallFollower f(c);
    Sense s = {-1.0f, -1.0f, false};
    bool was_bumped = false;
    double angle = 0, prev_a = std::atan2(w.y - 1.0, w.x - 1.5);
    double bumps = 0;
    for (int tick = 0; tick < 5000; ++tick) {
      uint32_t now = tick * 20;
      if (tick % 3 == 0) { s.front_m = w.range_m(); s.right_m = w.range_right_m(); }
      s.bump = w.bumped();
      Cmd cmd = f.update(s, now);
      w.drive(cmd.vl, cmd.vr);
      for (int k = 0; k < 20; ++k) w.step(0.001);
      bool b = w.bumped();
      if (b && !was_bumped) ++bumps;
      was_bumped = b;
      double a = std::atan2(w.y - 1.0, w.x - 1.5);
      double da = std::remainder(a - prev_a, 2 * SIM_PI);
      angle += da; prev_a = a;
      if (tick > 250 && w.x > 0 && w.x < 3 && w.y > 0 && w.y < 2) {
        double d = w.ray_from(w.x + 0.075 * std::sin(w.th), w.y - 0.075 * std::cos(w.th)
        double e = std::fmin(d, 1.0) - 0.30;
        sq += e * e; ++n;
      }
    }
    r.laps += angle / (2 * SIM_PI); r.bumps += bumps;
    if (full && f.state() == WallState::FAULT) r.faults += 1;
  }
  r.laps /= seeds; r.bumps /= seeds; r.rms_mm = std::sqrt(sq / n) * 1000; r.faults = 100
  return r;
}

int main() {
  std::printf("right-hand wall follower, 100 s, mean over 100 seeded runs\n");
```

```
  std::printf("  dropouts   controller   laps (counter-clockwise)   collisions per run
  for (double dp : {0.02, 0.30}) for (bool full : {false, true}) {
    R r = run(full, dp, 100);
    std::printf("  %6.0f %%   %-10s   %24.2f   %18.2f   %19.1f   %24.0f\n", dp * 100, fu
  }
  return 0;
}
```

```
right-hand wall follower, 100 s, mean over 100 seeded runs
  dropouts   controller   laps (counter-clockwise)   collisions per run   side error rms
       2 %   naive                            0.41                 1.00
       2 %   machine                          3.52                 0.00
      30 %   naive                            0.42                 1.00
      30 %   machine                          3.53                 0.00
```

The naive controller has one collision per run (the first corner) and gets only 0.41 of a lap; the machine goes round 3.5 times with no collisions and no FAULT, at both dropout rates. The state machine is what makes the difference. But look at the last column but one: the side error is about 100 mm *for both*, against a target of 300 mm, and that is large enough to be a finding, not a rounding. I first guessed it was the corners. A trace of the true side error and heading on the first wall, without noise or dropouts and then with them, says otherwise:

```
// d_wall.cpp -- diagnostic: the TRUE side error (mm) and heading (deg) every 0.5 s alon
#include <cstdio>
#include <cmath>
#include "sim.h"
#include "fsm.h"
static void trace(const char* name, double sigma, double dropout) {
  RobotParams rp; rp.tau = 0.1; SensorParams sp; sp.sigma_m = sigma; sp.dropout = dropou
  SimHW w(rp, sp, Arena(), 3); w.x = 0.5; w.y = 0.3; w.th = 0.0;
  WallConfig c = {0.3f, 0.30f, 4.0f, 0.40f, 0.70f, 1.5f, 0.15f, 0.8f, 0.15f, 300, 3000,
  WallFollower f(c); Sense s = {-1.0f, -1.0f, false};
  std::printf("%s\n  t(s):", name);
  for (int tick = 0; tick < 500; ++tick) {
    if (tick % 3 == 0) { s.front_m = w.range_m(); s.right_m = w.range_right_m(); }
    s.bump = w.bumped();
    Cmd cmd = f.update(s, tick * 20); w.drive(cmd.vl, cmd.vr);
    for (int k = 0; k < 20; ++k) w.step(0.001);
    if (tick % 25 == 24) { double d = w.ray_from(w.x + 0.075 * std::sin(w.th), w.y - 0.0
  }
  std::puts("");
```

```
}
int main() { trace("noise-free, no dropouts", 0.0, 0.0); trace("default noise and 2% dro
```

```
noise-free, no dropouts
  t(s):  -65(  +7)  -27( +15)   21( +17)   59( +14)   80(  +8)   86(  -1)   79( -10)   5
default noise and 2% dropouts
  t(s):  -64(  +8)  -23( +15)   27( +18)   66( +15)   85(  +7)   89(  -2)   80( -11)   5
```

Even with a perfect sensor the robot swings from −127 mm to +86 mm and back, with the heading going ±20°; the noisy trace is within about 12 mm of the clean one at every sample, so **the sensor is not the cause: the controller is**. Steering by a turn rate proportional to the distance error makes the heading the integral of the error and the lateral position the integral of the heading: a double integrator with a proportional gain is an oscillator with no damping, with period $2\pi/\sqrt{(v \cdot kp)} = 5.7$ s for $v = 0.3$ m/s and $kp = 4$. The upward crossings of the trace are 5.8 s apart (one seed, one trace; I would not claim the match to 0.1 s). Challenge 1 sweeps kp. The cure the theory gives is a term that damps (a heading or a rate-of-change term); I did not build or test one, so the chapter does not claim it works here.

7. Following a line

The line follower reads five reflectance sensors 80 mm ahead of the robot's centre, calibrates each one on a sweep across the line (Programming for Robotics with Arduino C++ 4), forms the weighted position of the line in millimetres, and steers on it. FOLLOW steers and slows down when the line is far off-centre; if no sensor sees the line it goes to SEARCH, spinning toward the side where the line was last seen, with the time-out of section 5; SEARCH ends in FAULT. The line here is a circle of radius 0.6 m with a response 25 mm either side; the score is the *true* distance of the robot's centre from the line, which only the simulator knows.

```
// e_line.cpp -- a line follower on a circular line (radius 0.6 m, 50 mm of response eit
// Machine: FOLLOW (steer on the weighted line position, slow down off the line), SEARCH
// 30 s, 20 seeded runs per row. Scored on the TRUE distance of the robot's centre from
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "fsm.h"
```

```
struct R { double rms_mm, lost, laps, faults; };

static R run(float v, float k_w, int seeds) {
  R r = {0, 0, 0, 0};
  double sq = 0; long n = 0;
  for (int seed = 1; seed <= seeds; ++seed) {
    RobotParams rp; rp.tau = 0.1;
    SimHW w(rp, SensorParams(), Arena(), seed);
    w.line.on = true;
    LineConfig c = {v, k_w, 0.5f, 1.5f, 0.15f, 3000};
    LineFollower f(c);
    w.x = w.line.cx + w.line.R; w.y = w.line.cy; w.th = SIM_PI / 2;                 // o
    int16_t raw[5];
    for (int i = -70; i <= 70; ++i) {                                                // 
      double x0 = w.x; w.x = x0 + i * 0.001; w.line_read(raw); f.cal().learn(raw); w.x =
    }
    double angle = 0, prev_a = std::atan2(w.y - w.line.cy, w.x - w.line.cx);
    for (int tick = 0; tick < 1500; ++tick) {
      w.line_read(raw);
      Cmd cmd = f.update(raw, tick * 20);
      w.drive(cmd.vl, cmd.vr);
      for (int k = 0; k < 20; ++k) w.step(0.001);
      double e = w.line_error(); sq += e * e; ++n;
      double a = std::atan2(w.y - w.line.cy, w.x - w.line.cx);
      angle += std::remainder(a - prev_a, 2 * SIM_PI); prev_a = a;
    }
    r.lost += f.lost_events(); r.laps += angle / (2 * SIM_PI);
    if (f.state() == LineState::FAULT) r.faults += 1;
  }
  r.rms_mm = std::sqrt(sq / n) * 1000; r.lost /= seeds; r.laps /= seeds; r.faults = 100.
  return r;
}

int main() {
  std::printf("line follower, 30 s, mean over 20 runs\n");
  std::printf("  speed (m/s)   steering gain (rad/s per mm)   rms distance from the line
  for (float v : {0.15f, 0.25f, 0.35f, 0.50f}) { R r = run(v, 0.05f, 20); std::printf("
  for (float k : {0.02f, 0.10f, 0.20f}) { R r = run(0.25f, k, 20); std::printf("  %11.2f
  return 0;
}
```

```
line follower, 30 s, mean over 20 runs
  speed (m/s)   steering gain (rad/s per mm)   rms distance from the line (mm)   times t
         0.15                          0.05                                1.5
```

```
0.25                    0.05                    1.2
0.35                    0.05                    5.3
0.50                    0.05                    9.9
0.25                    0.02                   11.8
0.25                    0.10                    2.0
0.25                    0.20                    3.6
```

Over this circle the follower never lost the line, not even at 0.50 m/s, the top speed of this robot; what changes with speed is the distance from the line, from about 1.2–1.5 mm at 0.15 and 0.25 m/s to 5.3 mm at 0.35 and 9.9 mm at 0.50 m/s. The 1.5 against 1.2 at the two slow speeds is a difference of 0.3 mm that I would not read as a trend. The gain matters more than the speed: at 0.25 m/s, 0.02 rad/s per mm leaves 11.8 mm (too weak to hold a curve), 0.05 gives 1.2, 0.10 gives 2.0 and 0.20 gives 3.6, which begins to ring. This is a favourable test: one smooth circle, with no gaps, sharp corners or crossings, a simulated array whose gains and offsets I made up, and a line wide enough for the array. A real track is harder, and the SEARCH and FAULT states exist for the part that I did not test on one.

8. On the Uno

`robot_ch7.ino` puts the table-driven avoider on the pin plan of Programming for Robotics with Arduino C++ 2: the non-blocking ultrasonic driver of the same chapter on pins 8 and 12, a median-of-five filter at the seam (a dropout passed on as a dropout), the bumper on pin 11, the motors through the dead-zone mapping of Programming for Robotics with Arduino C++ 5, and a 20 ms `think_task` on the scheduler of Programming for Robotics with Arduino C++ 1 that calls `machine.update()` and prints the state when it changes.

```
// robot_ch7.ino -- Programming for Robotics with Arduino C++, chapter 7: the obstacle a
// Compile-checked for the Uno (arduino:avr:uno). NOT run on a board: no board, motor, s
#include "scheduler.h"
#include "conv.h"
#include "echo.h"
#include "pins.h"
#include "filters.h"
#include "motion.h"
#include "fsm.h"

const float V_MAX_MPS = 0.5f;
const uint8_t DEAD_DUTY = 40;                          // ASSUMED: measure yours (Progra
```

```
static AvoidConfig CFG = {0.3f, 0.25f, 0.2f, 2.0f, 0.15f, 400, 400, 1200, 3, true};
static TableAvoider machine(CFG, 12345);
static Median<int16_t, 5> median;
static Echo echo;
static Scheduler<3> sch;
static bool trig_level = false;
static float g_range = -1.0f;                          // the filtered front distance; n
static uint8_t g_last_state = 255;

static void drive(float vl, float vr) {
  digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
  digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
  analogWrite(PIN_ML_PWM, speed_to_pwm_dz(vl >= 0.0f ? vl : -vl, V_MAX_MPS, DEAD_DUTY));
  analogWrite(PIN_MR_PWM, speed_to_pwm_dz(vr >= 0.0f ? vr : -vr, V_MAX_MPS, DEAD_DUTY));
}

static void echo_start_task(void*, uint32_t) { echo.start(); }

static void think_task(void*, uint32_t now) {          // every 20 ms: read, decide, act
  Sense s;
  s.front_m = g_range;
  s.right_m = -1.0f;
  s.bump = digitalRead(PIN_BUMP) == LOW;
  Cmd c = machine.update(s, now);
  drive(c.vl, c.vr);
  uint8_t st = machine.state();
  if (st != g_last_state) { g_last_state = st; Serial.print(now); Serial.print(F(" ms  s
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT);
  pinMode(PIN_ML_PWM, OUTPUT); pinMode(PIN_MR_PWM, OUTPUT);
  pinMode(PIN_TRIG, OUTPUT); pinMode(PIN_ECHO, INPUT);
  pinMode(PIN_BUMP, INPUT_PULLUP);
  sch.add(think_task, 0, 20);
  sch.add(echo_start_task, 0, 60);
}

void loop() {
  bool before = trig_level;
  Echo::Status st = echo.update(micros(), digitalRead(PIN_ECHO), trig_level);
  if (trig_level != before) digitalWrite(PIN_TRIG, trig_level);
  if (st == Echo::READY) g_range = (float)median.update((int16_t)(echo_us_to_m(echo.widt
```

```
    else if (st == Echo::TIMEOUT) g_range = -1.0f;      // a dropout is passed on as a dro
    sch.run(millis());
  }
```

It compiles for the Uno under `--warnings all` with no warning from the sketch to **7114 bytes of flash (22 %) and 381 bytes of RAM (18 %)**. For comparison, the earlier chapters' sketches were 8626 bytes and 433 bytes (Programming for Robotics with Arduino C++ 4), 8868 and 409 (Programming for Robotics with Arduino C++ 5) and 7638 and 541 (Programming for Robotics with Arduino C++ 6). It was *not run*: the avoider's gains (stop at 0.25 m, 400 ms backing, 400–1200 ms turning, a hold of 3 ticks) come from the simulator, the dead zone of 40 is an assumption, and how the real sensor reacts to a corner at a shallow angle is not in the simulator at all.

9. What the numbers can and cannot be trusted for

What they show. The shapes: the switch and the table give identical behaviour on 450 000 simulated ticks; the table is 474 bytes larger in flash for three states and flash tables save 22 bytes of RAM; a table can hold a bug (a shadowed row) that no simulation shows and a checker finds in a line; a missing exit or a mistyped state turns the robot into a spinner or a reverser; a machine without a time-out can spin for ever; the wall follower's error is the controller's own oscillation, not noise; the line follower holds a smooth circle to millimetres up to the robot's top speed. The sizes are the compiler's for the real chip.

What they do not show. The arena, the line, the sensor noise and the dropout rates, and the array's gains and offsets, are assumptions; a real room has furniture at angles, soft obstacles that return no echo, a floor that is not uniformly reflective. The collision counts are small numbers from a simulator whose obstacles are straight walls, and I would not use them to rank the two dropout rules. **No time per pass was measured** and nothing ran on a board. The wall follower's oscillation is diagnosed but not cured, and the line follower was not tried on a track with gaps, corners or crossings.

CHALLENGE 1: THE WALL FOLLOWER'S GAIN

Sweep the proportional gain kp of the wall follower (1, 2, 4, 8, 16 and 32 rad/s per metre) over 60 seeded runs of 100 s and report laps, collisions, the rms side error over the whole run, the rms side error in FOLLOW only (at least 2 s after the last corner) and the share of time in CORNER. Then trace the

true side error on the first wall without noise and compare the swing's period with $2\pi/\sqrt{(v \cdot kp)}$. Say what the sweep does and does not explain.

View solution

CHALLENGE 2: HOW LONG TO TRUST THE LAST ECHO

Sweep the avoider's `drop_limit` (1, 2, 3, 6, 10 and 20 control ticks) at 2 % and at 30 % dropouts and report collisions, arena coverage and the share of time not in CRUISE. Convert each limit to sensor readings, and explain the drop in coverage at 2 % dropouts for limits of 1 and 2.

View solution

CHALLENGE 3: HOW WIDE A CALIBRATION SWEEP

Sweep the line follower's calibration width (no sweep, and ±10, 20, 30, 40, 50 and 70 mm) and report the rms distance from the line, the number of times the line was lost, laps and the share of runs ending in FAULT. Explain the width below which calibration is wrong, and say why the rms of 0.0 mm for the run with no calibration is not good news.

View solution

QUICK REFERENCE

- **Switch machine.** `enum class` state in one byte, `switch` with every case; the behaviour is a pure `update(Sense, now) → Cmd`; the transition tick returns zero speeds.
- **Table machine.** Rows `{from, event, to, action}`, events as bits, the first matching row wins; tables in `PROGMEM` read with `pgm_read_byte` / `pgm_read_ptr`. Identical to the switch over 450 000 ticks.
- **Cost on the Uno.** Baseline 2158 B flash, 9 B RAM. Switch +1244 / +84. Table in RAM +1692 / +109. Table in flash +1718 / +87. Flash tables save 22 B of RAM (16 rows + 6 pointers) for 26 B more flash; the table is 474 B larger than the switch at three states.
- **Checker.** Out-of-range rows, duplicate (state, event), unreachable states, states that cannot return. Missing TURN→CRUISE: 1.4 % coverage. Mistyped BACKUP row: 3.2 %, one collision per run. Duplicate row: no visible difference in simulation, one line from the checker.
- **Time-out.** Every state that waits on the world needs a clock exit and a FAULT: 3 s gives FAULT from 4 s on (0.7 revolutions); 65.5 s spins 7.1 revolutions in 30 s.

- **Avoider.** `drop_limit` counts 20 ms ticks, 3 = one 60 ms reading. 30 % dropouts: naive 22.4 % coverage, safe hold 3 8.1 %, hold 6 16.5 %, hold 10 20.6 %; collisions 0.00–0.10 per run.
- **Wall follower.** Naive: 1.00 collision, 0.41 laps. Machine: 3.52 laps, 0 collisions, 0 FAULT; side error about 100 mm is an oscillation of the P controller (period near $2\pi/\sqrt{(v \cdot kp)} = 5.7$ s), not noise. Damping term not built.
- **Line follower.** Circle R 0.6 m: never lost up to 0.50 m/s; rms 1.2 mm at 0.25 m/s, 9.9 mm at 0.50; gain 0.05 best of four at 0.25 m/s. Calibration sweep needs at least the outer sensor offset (40 mm); with none, FAULT in every run.
- **Budget.** `robot_ch7.ino`: 7114 bytes flash (22 %), 381 bytes RAM (18 %), compiled, not run.
- **Not run:** anything on a board; time per pass; a track with gaps or crossings; a cure for the wall follower's oscillation.

Next: Programming for Robotics with Arduino C++ 8, Concurrency and Real-Time Basics

Programming for Robotics with Arduino C++ 8 (planned; not yet written) looks at what happens when more than one thing wants the processor: interrupts and shared data, what a long task does to a short one, priorities in a cooperative scheduler, and how to find out whether the robot of this chapter still meets its 20 ms deadline once the sensors, the controllers and the machine all run together.

Programming for Robotics with Arduino C++

Course 2 · Chapter 8 · Concurrency and Real-Time Basics

Programming for Robotics with Arduino C++ 7 gave the robot decisions, and every one of them assumed that `update()` would be called on time and that nothing else would get in the way. This chapter looks at what else is happening on the chip. The processor does one thing at a time, but it can be *interrupted*: a pin changes or a timer reaches a number, the hardware stops `loop()` in the middle of an instruction, runs a short function, and carries on. That is the nearest thing to concurrency an Uno has, and it brings the oldest bugs in embedded software: a variable the compiler thinks cannot change, a 32-bit number read half-way through an update, a long task that makes a short one late, a robot that hangs.

The chapter has the course's usual habit: where it can, the claim is shown by the compiler or by a run. The disassembly of the avr-gcc compiler shows what `volatile` does; a simulation shows what a long `Serial` dump does to a 20 ms control loop, and that my own first prediction of it was wrong; a model of the watchdog shows that most of the places you could feed it do not catch the faults you wanted it for.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Compiled for the Uno, and the machine code read** (arduino-cli, core `arduino:avr` 1.8.8, `avr-gcc` 7.3, `avr-objdump`): the `volatile` and atomic-read examples, and `robot_ch8.ino` with its Timer1 interrupt. The instruction listings are the compiler's own. **Run on a PC** (g++ 16.2, C++11): the timer-setting calculator, the polling model, the scheduler on a *virtual* clock, and a model of the watchdog. **The task times in the scheduler experiment are assumptions I chose**; the watchdog is a software model of one, not the chip's. **Not done**: nothing was attached to a board. **No interrupt latency, no execution time and no watchdog reset was measured on hardware**, so the chapter counts instructions and says what they are, never microseconds of real machine time.

1. What an interrupt is, in C++ for a Python programmer

An **interrupt service routine** (ISR) is a function the hardware calls when an event happens, with no call in your code: `ISR(INT0_vect) { ... }` runs when pin 2 changes (the encoder, Programming for Robotics with Arduino C++ 4), `ISR(TIMER1_COMPA_vect)` when Timer1 reaches its compare value. While it runs, further interrupts are held off. It then returns to exactly where `loop()` was.

IN PYTHON	ON THE UNO IN C++
`threading.Thread`, `asyncio`: the runtime switches between tasks	there is one thread; the only preemption is an ISR, and only for as long as the ISR runs
a lock (`threading.Lock`) protects shared data	shared data is protected by `volatile` and by switching interrupts off for a few instructions (section 2)
`time.sleep(0.02)` lets other threads run	`delay()` would stop the robot; time is checked against a counter (Programming for Robotics with Arduino C++ 1)
an exception, a traceback and a process that ends	a hang: the program just stops, unless a watchdog resets it (section 6)
the language runtime decides when a task runs	you decide: the ISR is hardware-timed, `loop()` is as late as its slowest task (section 5)

Four rules keep an ISR from doing harm; they are conventions rather than things the compiler checks. **Keep it short**: nothing else interrupts it, and the time it takes is added to every other task's lateness. **Do not call `delay()` or `Serial.print()` in it**: `millis()` does not advance inside an ISR, and `Serial` uses interrupts itself. **Do not do real work in it**: count, copy, set a flag, and let `loop()` do the rest. **Everything it shares with `loop()` is `volatile`**, which is the next section.

2. `volatile` and atomic access, as the compiler sees them

Two bugs are the reason for the second and third rules, and the chapter does not ask you to take them on trust. The program below has a flag set by an ISR and a 32-bit counter it increments, and four functions: one that waits for the flag, one that reads the counter plainly, one that reads it inside `ATOMIC_BLOCK`. It was compiled twice for the Uno, once with `volatile` on the two variables and once without, and the machine code of the three functions was read with `avr-objdump`.

```
// s_vol.ino -- what `volatile` and an atomic block change in the machine code (Programm
// VOL is `volatile` or nothing (-DVOL=). ATOMIC selects the guarded copy of a 32-bit co
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>

#ifndef VOL
```

```
#define VOL volatile
#endif

VOL bool g_flag = false;
VOL uint32_t g_ticks = 0;

ISR(INT0_vect) { g_flag = true; g_ticks++; }

__attribute__((noinline)) void wait_flag() { while (!g_flag) { } }

__attribute__((noinline)) uint32_t read_plain() { return g_ticks; }

__attribute__((noinline)) uint32_t read_atomic() {
  uint32_t t;
  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { t = g_ticks; }
  return t;
}

void setup() { pinMode(13, OUTPUT); EICRA = _BV(ISC00); EIMSK = _BV(INT0); }
void loop() {
  wait_flag();
  g_flag = false;
  uint32_t a = read_plain(), b = read_atomic();
  digitalWrite(13, (a ^ b) & 1);
}
```

```
Sketch uses 894 bytes (2%) of program storage space. Maximum is 32256 bytes.
=== volatile
000000e0 <read_atomic()>:
  e0:   2f b7           in      r18, 0x3f       ; 63
  e2:   f8 94           cli
  e4:   60 91 00 01     lds     r22, 0x0100     ; 0x800100 <_edata>
  ec:   80 91 02 01     lds     r24, 0x0102     ; 0x800102 <_edata+0x2>
  f0:   90 91 03 01     lds     r25, 0x0103     ; 0x800103 <_edata+0x3>
  f4:   2f bf           out     0x3f, r18       ; 63
  f6:   08 95           ret

000000f8 <read_plain()>:
  f8:   60 91 00 01     lds     r22, 0x0100     ; 0x800100 <_edata>
  fc:   70 91 01 01     lds     r23, 0x0101     ; 0x800101 <_edata+0x1>
 100:   80 91 02 01     lds     r24, 0x0102     ; 0x800102 <_edata+0x2>
 104:   90 91 03 01     lds     r25, 0x0103     ; 0x800103 <_edata+0x3>
```

```
 108:   08 95           ret

0000010a <wait_flag()>:
 10a:   80 91 04 01     lds     r24, 0x0104     ; 0x800104 <g_flag>
 10e:   88 23           and     r24, r24
 110:   e1 f3           breq    .-8             ; 0x10a <wait_flag()>
 112:   08 95           ret

Sketch uses 896 bytes (2%) of program storage space. Maximum is 32256 bytes.
=== plain
000000e0 <read_atomic()>:
  e0:   2f b7           in      r18, 0x3f       ; 63
  e2:   f8 94           cli
  e4:   60 91 00 01     lds     r22, 0x0100     ; 0x800100 <_edata>
  e8:   70 91 01 01     lds     r23, 0x0101     ; 0x800101 <_edata+0x1>
  ec:   80 91 02 01     lds     r24, 0x0102     ; 0x800102 <_edata+0x2>
  f0:   90 91 03 01     lds     r25, 0x0103     ; 0x800103 <_edata+0x3>
  f4:   2f bf           out     0x3f, r18       ; 63
  f6:   08 95           ret

000000f8 <read_plain()>:
  f8:   60 91 00 01     lds     r22, 0x0100     ; 0x800100 <_edata>
  fc:   70 91 01 01     lds     r23, 0x0101     ; 0x800101 <_edata+0x1>
 100:   80 91 02 01     lds     r24, 0x0102     ; 0x800102 <_edata+0x2>
 104:   90 91 03 01     lds     r25, 0x0103     ; 0x800103 <_edata+0x3>
 108:   08 95           ret

0000010a <wait_flag()>:
 10a:   80 91 04 01     lds     r24, 0x0104     ; 0x800104 <g_flag>
 10e:   81 11           cpse    r24, r1
 110:   01 c0           rjmp    .+2             ; 0x114 <wait_flag()+0xa>
 112:   ff cf           rjmp    .-2             ; 0x112 <wait_flag()+0x8>
 114:   08 95           ret
```

The flag. Look at `wait_flag()`. With `volatile` (first listing) it loads the flag from memory, tests it, and branches back to the load: `lds`, `and`, `breq .-8`. Without it (second listing) the compiler reasons that nothing in this loop changes `g_flag`, so it needs to load it only once: `lds`, `cpse` (skip the next instruction if it is zero), and then either return or `rjmp .-2`, **a jump to itself, for ever**. The interrupt that sets the flag can run all it likes; the loop does not look. The compiler did nothing wrong: the language says a variable that nothing in the program changes does not change. `volatile` is how you say "something outside the program does", and it makes no difference to the size here (894 bytes for the whole sketch with it, 896 without).

The counter. Look at `read_plain()` and `read_atomic()`. A 32-bit value on an 8-bit chip is four separate loads (`lds r22`, `r23`, `r24`, `r25`). An interrupt that fires between two of them and increments the counter leaves the copy with some bytes from before and some from after: when a byte carries (0x00FF to 0x0100) the result can be off by 256. That is the torn read of Programming for Robotics with Arduino C++ 4, now seen in the machine code. `read_atomic()` saves the status register, executes `cli` (interrupts off), loads the four bytes, and restores the status register with `out 0x3f`: eight instructions where the plain read has five, and no interrupt can arrive in the middle. `ATOMIC_RESTORESTATE` restores whatever the interrupt-enable bit was, so it is safe to call from code that already had interrupts off. Both listings are identical for the two builds, as they should be: `volatile` does not make a 32-bit read atomic, and an atomic block does not make a variable `volatile`. A single byte read (the flag) is one instruction and needs no guard; a 16-bit value is two instructions and does.

3. Hardware timers

The Uno's ATmega328P has three timers: Timer0 (8 bit) and Timer2 (8 bit), and Timer1 (16 bit). The Arduino core already uses Timer0 for `millis()`, `micros()` and PWM on pins 5 and 6, which are the build robot's motor speed pins (Programming for Robotics with Arduino C++ 2), so changing it would break both the clock and the motors. Timer2 drives PWM on pins 3 and 11, and Timer1 on pins 9 and 10, which the pin plan does not use, so Timer1 is the one this chapter takes. (The `Servo` library also takes Timer1 on the Uno; that is a documented conflict I did not test here.)

In **CTC mode** (clear timer on compare) a timer counts from 0 to a value `OCR` and starts again, firing an interrupt at each match, so the frequency is `F_CPU / (prescaler × (OCR + 1))`. The prescaler divides the 16 MHz clock by 1, 8, 64, 256 or 1024 (Timer2 also offers 32 and 128), and `OCR` is 8 or 16 bits wide. The two limits together decide which frequencies are possible and how accurate they are. The calculator is plain C++, so a PC runs it:

```
// timer_calc.h -- choosing a prescaler and a compare value for an AVR timer in CTC mode
// In CTC mode the timer counts 0..OCR and starts again, so the compare-match frequency
#pragma once
#include <stdint.h>
#include <cmath>

struct TimerPlan { bool ok; uint16_t prescaler; uint32_t ocr; double actual_hz; double e
```

```
// bits: 8 for Timer0/Timer2 (OCR 0..255), 16 for Timer1 (OCR 0..65535). `prescalers` li
inline TimerPlan plan_timer(double target_hz, const uint16_t* prescalers, int n, int bit
  TimerPlan best = {false, 0, 0, 0.0, 1e18};
  uint32_t max_ocr = bits == 8 ? 255u : 65535u;
  for (int i = 0; i < n; ++i) {
    double ticks = f_cpu / (prescalers[i] * target_hz);          // timer counts per per
    double ocr_real = ticks - 1.0;
    if (ocr_real < 0.0 || ocr_real > (double)max_ocr + 0.5) continue;
    uint32_t ocr = (uint32_t)std::floor(ocr_real + 0.5);
    if (ocr > max_ocr) ocr = max_ocr;
    double actual = f_cpu / (prescalers[i] * (double)(ocr + 1));
    double err = std::fabs(actual - target_hz) / target_hz * 1e6;
    if (err < best.error_ppm) { best.ok = true; best.prescaler = prescalers[i]; best.ocr
  }
  return best;
}
```

```
// e_timer.cpp -- choosing timer settings on the host. For several wanted frequencies it
// with the smallest frequency error, and prints the frequency really produced. Then the
#include <cstdio>
#include "timer_calc.h"

int main() {
  const uint16_t P1[] = {1, 8, 64, 256, 1024};                  // Timer1 and Timer0
  const uint16_t P2[] = {1, 8, 32, 64, 128, 256, 1024};         // Timer2
  const double targets[] = {1.0, 50.0, 440.0, 1000.0, 20000.0, 38000.0, 100000.0};
  std::printf("CTC settings at 16 MHz: best prescaler and OCR per timer, and the frequen
  std::printf("  wanted (Hz)   Timer1 (16 bit): presc  OCR     actual (Hz)   error (ppm)
  for (double t : targets) {
    TimerPlan a = plan_timer(t, P1, 5, 16), b = plan_timer(t, P2, 7, 8);
    std::printf("  %11.0f   ", t);
    if (a.ok) std::printf("%21u  %6lu  %12.3f  %12.1f   ", a.prescaler, (unsigned long)a
    if (b.ok) std::printf("%20u  %4lu  %12.3f  %12.1f\n", b.prescaler, (unsigned long)b.
  }
  std::printf("\nTimer0 as the Arduino core sets it up (prescaler 64, 8-bit overflow):\n
  double tick_us = 64.0 / 16.0, overflow_us = 256.0 * tick_us;
  std::printf("  one timer count      = %.1f us  (so micros() moves in steps of 4 us)\n"
  std::printf("  one overflow         = %.1f us  (the millis() interrupt fires at %.3f H
  // the core's own bookkeeping (wiring.c, arduino:avr 1.8.8): 1 ms per overflow, and th
  const int MILLIS_INC = 1024 / 1000, FRACT_INC = (1024 % 1000) >> 3, FRACT_MAX = 1000 >
  unsigned long ms = 0; int f = 0; unsigned long n = 0;
  while (ms < 1000000UL) { ms += MILLIS_INC; f += FRACT_INC; if (f >= FRACT_MAX) { f -= F
  std::printf("  the core adds %d ms and %d (x 8 us) of fraction per overflow, and a spa
```

```
    std::printf("  simulated: %lu overflows (%.3f s of real time) are counted as %lu ms; e
    return 0;
}
```

```
CTC settings at 16 MHz: best prescaler and OCR per timer, and the frequency really produ
  wanted (Hz)   Timer1 (16 bit): presc  OCR     actual (Hz)   error (ppm)   Timer2 (8 bi
            1                      256   62499         1.000           0.0   out of range
           50                        8   39999        50.000           0.0   out of range
          440                        1   36363       439.996          10.0
         1000                        1   15999      1000.000           0.0
        20000                        1     799     20000.000           0.0
        38000                        1     420     38004.751         125.0
       100000                        1     159    100000.000           0.0

Timer0 as the Arduino core sets it up (prescaler 64, 8-bit overflow):
  one timer count      = 4.0 us  (so micros() moves in steps of 4 us)
  one overflow         = 1024.0 us  (the millis() interrupt fires at 976.562 Hz)
  the core adds 1 ms and 3 (x 8 us) of fraction per overflow, and a spare ms when the fr
  simulated: 976563 overflows (1000.001 s of real time) are counted as 1000000 ms; error
```

Read the table for what the limits mean. A 1 kHz tick is exact on both timers (Timer1: prescaler 1, `OCR1A` = 15999; Timer2: prescaler 64, `OCR2A` = 249). 50 Hz and 1 Hz need Timer1: an 8-bit timer cannot count slowly enough, since the lowest it can reach is about 61 Hz. 440 Hz (a musical A) is 10 ppm off on Timer1 and 320 ppm off on Timer2, which is 0.02 of a cent against 0.55, and the 38 kHz of an infrared remote carrier is 125 ppm off on Timer1 and nearly 7000 ppm (0.7 %) off on Timer2, where the 8-bit compare value is too coarse. The last lines are the Arduino core's own bookkeeping for Timer0, which I read from its source and ran: one timer count is 4 µs, which is why `micros()` moves in steps of 4 µs; an overflow takes 1.024 ms, not 1 ms, and the core keeps the extra 24 µs as a fraction and adds a spare millisecond when it reaches 125 (units of 8 µs), so `millis()` stays within about 1 ppm of the *timer* (the simulated run printed above counts 1000.001 s of timer time as 1000000 ms). The timer, of course, is only as accurate as the board's clock, whose error I did not measure.

Challenge 1 asks how many whole frequencies each timer can reach. The setup of Timer1 for a 1 kHz tick is five register writes (in `timer1_begin_1khz()` in section 8).

4. Polling against events

The alternative to an interrupt is to **poll**: read the pin every time round `loop()`. It is simpler and has no shared variables, and it works when the signal is slow compared with the loop. When it is not, polling misses events. The experiment models a digital input that gets pulses of width *w* at random moments (about 20 per second, never overlapping), a polling loop that reads the pin every *T* and counts a pulse when it sees the pin high after seeing it low, and an interrupt that counts every rising edge (the reference: the model assumes its latency is much shorter than the pulse, which I did not measure):

```
// e_poll.cpp -- polling against an interrupt, as a model (host only; nothing here ran o
// A digital input gets pulses of width w at random moments (about 20 per second, never
// the pin high after seeing it low. An interrupt counts every rising edge, so it is the
// For each (w, T): the share of pulses the polling loop sees, over 20 seeds of 100 s, a
#include <cstdio>
#include <cmath>
#include <vector>
#include <initializer_list>
#include "sim.h"

struct Result { double seen, formula; long pulses; };

static Result run(double w, double T, int seeds) {
  long pulses = 0, seen = 0;
  for (int seed = 1; seed <= seeds; ++seed) {
    Rng rng((uint32_t)seed * 7919u);
    std::vector<double> start;
    double t = 0.05;
    while (t < 100.0) { start.push_back(t); t += w + (-std::log(1.0 - rng.uniform() * 0.
    double phase = rng.uniform() * T;
    bool prev = false; size_t k = 0;
    long s = 0;
    for (double ts = phase; ts < 100.0; ts += T) {
      while (k < start.size() && start[k] + w < ts) ++k;                    // pulses
      bool high = k < start.size() && start[k] <= ts && ts <= start[k] + w;
      if (high && !prev) ++s;
      prev = high;
    }
    pulses += (long)start.size(); seen += s;
  }
  Result r = {(double)seen / (double)pulses, w / T < 1.0 ? w / T : 1.0, pulses};
  return r;
}

int main() {
```

```
  std::printf("share of pulses a polling loop sees (an interrupt sees 100 %%, in this mo
  std::printf("  loop period T:        0.1 ms          1 ms          5 ms         20 ms\
  for (double w : {0.0002, 0.001, 0.005, 0.02, 0.1}) {
    std::printf("  pulse %6.1f ms:", w * 1000);
    for (double T : {0.0001, 0.001, 0.005, 0.02}) { Result r = run(w, T, 20); std::print
    std::printf("\n");
  }
  std::printf("(in brackets: the formula min(1, w / T))\n");
  return 0;
}
```

```
share of pulses a polling loop sees (an interrupt sees 100 %, in this model), 20 pulses/
  loop period T:        0.1 ms          1 ms          5 ms         20 ms
  pulse    0.2 ms:    99.9% (100%)    20.0% ( 20%)     3.9% (  4%)     1.0% (  1%)
  pulse    1.0 ms:    99.9% (100%)    99.0% (100%)    19.8% ( 20%)     4.9% (  5%)
  pulse    5.0 ms:    99.9% (100%)    99.0% (100%)    95.3% (100%)    22.8% ( 25%)
  pulse   20.0 ms:    99.9% (100%)    99.0% (100%)    95.3% (100%)    82.6% (100%)
  pulse  100.0 ms:    99.9% (100%)    99.0% (100%)    95.2% (100%)    82.5% (100%)
(in brackets: the formula min(1, w / T))
```

The share a polling loop sees is close to the formula `min(1, w/T)` when the pulse is shorter than the loop period: a 0.2 ms pulse polled every 1 ms is seen 20 % of the time, every 5 ms 3.9 %, every 20 ms 1.0 %. The odd entries are on the right of the table, where the formula says 100 % and the model says less (95.3 %, 82.6 %): when *w* is longer than *T*, polling sees every pulse that is separated from the next by more than one polling period, but two pulses closer together than that look like one, and with an average gap of 50 ms that happens to one pulse in six at a 20 ms period. For the build robot's encoder (Programming for Robotics with Arduino C++ 4) the pulse is the width of one slot at the speed of the wheel, a fraction of a millisecond at speed, and a control loop that runs every 20 ms would see almost none of them; that is why encoders use interrupts and the bumper switch, whose press lasts tens of milliseconds, can be polled. The rule that follows is one of timing, not of taste: **poll when the thing you are waiting for lasts several loop periods, use an interrupt when it can be shorter than one**.

5. A cooperative scheduler, and what a long task does to it

The scheduler of Programming for Robotics with Arduino C++ 1 is *cooperative*: each task runs to its end, and then the next due task starts. It has no preemption, so the control task waits for

whatever is running when it falls due, whatever its priority. The experiment puts a control task (20 ms period, assumed 1.2 ms of work) and an echo task (60 ms) on the real `Scheduler` class with a virtual clock, adds a report task whose time varies (the cost of writing text to `Serial` when the transmit buffer is full: 10 bit-times per character, 86.8 µs at 115200 baud and 1042 µs at 9600), and records the control task's worst lateness. My prediction: *about the length of the longest other task*. The first run, with the report period at 500 ms, said otherwise:

```
// e_sched.cpp -- what a long task does to a short one in a COOPERATIVE scheduler (the S
// Task execution times are ASSUMED, not measured: control 1.2 ms every 20 ms, echo star
// A task cannot be interrupted by another task, so the control task, however high its p
// Serial text takes 10 bit-times per character (8 data + start + stop): 86.8 us per cha
#include <cstdio>
#include <cstdint>
#include <initializer_list>
#include "scheduler.h"

static uint32_t g_now = 0;                              // virtual microseconds
struct Ctx { uint32_t cost_us; };

static void run_task(void* c, uint32_t) { g_now += static_cast<Ctx*>(c)->cost_us; }

struct Row { const char* name; uint32_t report_us; };

int main() {
  const Row rows[] = {
    {"no report task", 0},
    {"report 40 chars at 115200 baud (3.5 ms)", 3472},
    {"report 40 chars at 9600 baud (41.7 ms)", 41667},
    {"dump 200 chars at 115200 baud (17.4 ms)", 17360},
    {"dump 200 chars at 9600 baud (208 ms)", 208333},
    {"the 200-char dump in 5 pieces of 40 chars at 115200 baud (3.5 ms each)", 3472},
  };
  std::printf("cooperative scheduler, 60 s of virtual time: control task every 20 ms (1.
  for (int mode = 0; mode < 2; ++mode) {
  std::printf("\n%s\n", mode == 0 ? "report period 500 ms (a multiple of the control per
  std::printf("  case
  for (int i = 0; i < 6; ++i) {
    g_now = 0;
    Ctx control = {1200}, echo = {100}, report = {rows[i].report_us};
    Scheduler<4> sch;
    sch.add(run_task, &control, 20000);
    sch.add(run_task, &echo, 60000);
    if (rows[i].report_us) sch.add(run_task, &report, i == 5 ? (mode ? 97000u : 100000u)
```

```
    while (g_now < 60000000u) { sch.run(g_now); g_now += 10; }            // each pass
    const Task& c = sch.task(0);
    std::printf("  %-72s   %27.2f   %20lu   %12lu\n", rows[i].name, c.worst_late / 1000.
  }
  }
  std::printf("(prediction: worst lateness is about the longest OTHER task: 0, 3.5, 41.7
  return 0;
}
```

```
cooperative scheduler, 60 s of virtual time: control task every 20 ms (1.2 ms of work)

report period 500 ms (a multiple of the control period: the two are always due together)
  case                                                                    worst contr
  no report task
  report 40 chars at 115200 baud (3.5 ms)
  report 40 chars at 9600 baud (41.7 ms)
  dump 200 chars at 115200 baud (17.4 ms)
  dump 200 chars at 9600 baud (208 ms)
  the 200-char dump in 5 pieces of 40 chars at 115200 baud (3.5 ms each)

report period 497 ms (3 ms off the grid each time: over 60 s it lands at every phase aga
  case                                                                    worst contr
  no report task
  report 40 chars at 115200 baud (3.5 ms)
  report 40 chars at 9600 baud (41.7 ms)
  dump 200 chars at 115200 baud (17.4 ms)
  dump 200 chars at 9600 baud (208 ms)
  the 200-char dump in 5 pieces of 40 chars at 115200 baud (3.5 ms each)
(prediction: worst lateness is about the longest OTHER task: 0, 3.5, 41.7, 17.4, 208 and
```

In the first table the prediction fails: a 3.5 ms report gives 0.01 ms of lateness, a 17.4 ms dump 0.00. The reason is in the heading I added afterwards: 500 ms is a multiple of 20 ms, so the report and the control task fall due *at the same moment* and, with the control task first in the table, always run in the order control, report. The report never gets in the way, so the experiment had been measuring a coincidence. Only the 41.7 and 208 ms cases, which are longer than a control period, show damage (23 and 190 ms, with 120 and 1080 control runs skipped). The second table moves the report off the grid (497 ms, so over 60 s it falls at every phase against the control task), and the prediction holds: the worst lateness is 2.49 ms for the 3.5 ms report, 16.37 for the 17.4 ms dump, 40.69 for 41.7 and 207.34 for 208, each about 1 ms below the length of the task because the phases are 1 ms apart. And the cure is as the last row

shows: **the same 200 characters in five 40-character pieces** has the same worst case as one piece (2.49 ms) instead of 16.37 ms.

Two lessons. First, the worst case of a cooperative scheduler is **the longest task that can be running when a short one falls due**, so the number to design for is the longest task, not the average load. Second, a test that happens to align the tasks measures nothing about the worst case: the check in the second table (a period that is off the grid) is as important as the first. The fixes are to split long work into pieces that fit (challenge 2 finds the size), to let the hardware do the waiting (the Arduino `Serial` has a 64-byte transmit buffer and `print()` returns at once while there is room; I did not test the Uno's buffer limit here), and to give the time-critical work to a timer interrupt, as `robot_ch8.ino` does for its tick.

6. The watchdog

A **watchdog timer** is a hardware counter that resets the chip when it reaches its timeout unless the program resets it first (`wdt_reset()`). It is the answer to "what if the program hangs?", which on a robot with live motors is not a theoretical question. It is also easy to defeat, because the question is *where the program feeds it*. `wdt.h` holds a model of the hardware (for tests) and a `Heartbeats` class: every task that matters calls `checkin(id)` when it has done its work, and the watchdog is fed only when all of them have.

```
// wdt.h -- the logic around a watchdog timer (Programming for Robotics with Arduino C++
// The hardware watchdog resets the chip if it is not "fed" (wdt_reset()) within its tim
#pragma once
#include <stdint.h>

// A model of the hardware: it counts down and resets when it reaches the timeout.
class WdtModel {
 public:
  explicit WdtModel(uint32_t timeout_ms) : timeout_(timeout_ms), last_feed_(0), reset_at
  void feed(uint32_t now) { if (!reset_) last_feed_ = now; }
  bool poll(uint32_t now) { if (!reset_ && now - last_feed_ >= timeout_) { reset_ = true
  bool has_reset() const { return reset_; }
  uint32_t reset_at() const { return reset_at_; }
 private:
  uint32_t timeout_, last_feed_, reset_at_;
  bool reset_;
};
```

```
// Heartbeats: every task that matters calls checkin(id) when it has done its work. The w
template <uint8_t N>
class Heartbeats {
 public:
  Heartbeats() : seen_(0) {}
  void checkin(uint8_t id) { seen_ |= (uint8_t)(1u << id); }
  bool all_alive() const { return seen_ == (uint8_t)((1u << N) - 1u); }
  void clear() { seen_ = 0; }
  uint8_t mask() const { return seen_; }
 private:
  uint8_t seen_;
};
```

The experiment runs three feeding policies against four situations, with a 250 ms timeout and a fault injected at 1 s: feed on every pass of `loop()`; feed only when both tasks have checked in; feed from a 1 ms timer interrupt.

```
// e_wdt.cpp -- where should the watchdog be fed? Three feeding policies against four si
// The robot: a control task every 20 ms and an echo task every 60 ms, each checking in w
//   loop hangs      the main loop stops completely (an infinite loop somewhere)
//   control dies    the control task silently stops running, but loop() keeps spinning
//   echo dies       the echo task silently stops running, but loop() keeps spinning
//   none            control run: nothing goes wrong (any reset here is a FALSE reset)
// Policies: "in loop": wdt feed on every pass of loop().  "supervisor": feed only when b
// Result: the time from the fault to the reset in ms, or "never" (nothing reset the robo
#include <cstdio>
#include "wdt.h"

enum Fault { NONE, LOOP_HANGS, CONTROL_DIES, ECHO_DIES };
enum Policy { IN_LOOP, SUPERVISOR, IN_ISR };

static long run(Policy pol, Fault fault, uint32_t timeout_ms, uint32_t horizon_ms) {
  WdtModel wdt(timeout_ms);
  Heartbeats<2> hb;
  const uint32_t t_fault = 1000;
  for (uint32_t t = 1; t <= horizon_ms; ++t) {
    bool loop_alive = !(fault == LOOP_HANGS && t >= t_fault);
    bool control_alive = !(fault == CONTROL_DIES && t >= t_fault);
    bool echo_alive = !(fault == ECHO_DIES && t >= t_fault);
    if (loop_alive) {
      if (t % 20 == 0 && control_alive) hb.checkin(0);
      if (t % 60 == 0 && echo_alive) hb.checkin(1);
```

```
      if (pol == IN_LOOP) wdt.feed(t);
      if (pol == SUPERVISOR && hb.all_alive()) { wdt.feed(t); hb.clear(); }
    }
    if (pol == IN_ISR) wdt.feed(t);                    // the interrupt fires whether o
    if (wdt.poll(t)) return fault == NONE ? -2 : (long)(wdt.reset_at() - t_fault);
  }
  return -1;
}

static void cell(long r) {
  if (r == -1) std::printf("  %14s", "never");
  else if (r == -2) std::printf("  %14s", "FALSE RESET");
  else std::printf("  %11ld ms", r);
}

int main() {
  std::printf("watchdog timeout 250 ms: time from the fault to the reset (never = not ca
  std::printf("  situation       feed in loop   supervisor     feed in timer ISR\n");
  const char* names[] = {"none", "loop hangs", "control dies", "echo dies"};
  for (int f = 0; f < 4; ++f) {
    std::printf("  %-12s", names[f]);
    for (int p = 0; p < 3; ++p) cell(run((Policy)p, (Fault)f, 250, 5000));
    std::printf("\n");
  }
  return 0;
}
```

```
watchdog timeout 250 ms: time from the fault to the reset (never = not caught within 5 s
  situation       feed in loop   supervisor     feed in timer ISR
  none                   never          never          never
  loop hangs           249 ms         210 ms          never
  control dies           never          270 ms          never
  echo dies              never          210 ms          never
```

Read the table by columns. **Feeding in** `loop()` catches only a total hang (249 ms): if the control task silently stops while `loop()` spins on, the robot carries on, with its last command, un-watched. **Feeding from a timer interrupt catches nothing**, not even the total hang, because the interrupt keeps firing while `loop()` is stuck: it proves that the timer works. It is the most common watchdog mistake, and it looks like safety. **The supervisor** catches all three, after 210 to 270 ms; the 270 ms is the timeout plus up to one heartbeat period (60 ms), which is the cost of feeding only on a full set of check-ins. The first row, “none”, is the control: with no fault there

is no reset in 5 s with any policy. The choice of the timeout is the subject of challenge 3: too short and a healthy robot is reset by a long but harmless task.

On the Uno the calls are `wdt_enable(WDTO_250MS)` and `wdt_reset()` from `<avr/wdt.h>`; `robot_ch8.ino` compiles them in. Whether a watchdog reset works cleanly on *your* board depends on its bootloader (the Uno R3's is expected to cope, but old bootloaders can loop after a watchdog reset), and how to find out afterwards *why* the chip reset depends on it too. I did not test either on a board.

7. Why there are no threads here, and what an RTOS would add

A **real-time operating system** (RTOS) gives each task its own stack and switches between them, usually on a timer interrupt, so a high-priority task can *preempt* a low-priority one in the middle. It would remove the section 5 problem: a control task could preempt a serial dump. It would also cost: each task needs its own stack (a stack of a hundred bytes or so is a realistic minimum for even small code, and it must be big enough for the deepest call and for an ISR on top; several tasks at that size are a large share of the Uno's 2048 bytes, of which `robot_ch8.ino` already uses 327), the scheduler itself takes flash and RAM, and every variable shared between tasks now needs a lock, where today only the few that an ISR touches do. I did **not** build or measure an RTOS for the Uno, so those are descriptions of the cost, not figures. The sensible reading for this robot: the cooperative scheduler plus a few short ISRs gives deadline control that can be *analysed* (section 5's rule), in 6918 bytes. An RTOS pays off when there are several tasks that must preempt each other, and on a larger board (the Pico, the ESP32), where the RAM is not the constraint.

8. On the Uno

`robot_ch8.ino` is chapter 7's avoider rebuilt around a timer. Timer1 fires every millisecond; its ISR keeps a millisecond count, sets a flag every 20 ms for the control task and every 60 ms for the echo, and counts a *control overrun* if the control flag is still set when the next one comes (a measure of how often `loop()` is too slow). The encoders count on their own interrupts. `loop()` serves the flags, checks both tasks in with the heartbeat, and feeds the watchdog only when both have run.

```
// robot_ch8.ino -- Programming for Robotics with Arduino C++, chapter 8: chapter 7's av
// and a watchdog fed only when every task has checked in. Compile-checked for the Uno (
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <util/atomic.h>
#include "conv.h"
#include "echo.h"
#include "pins.h"
#include "filters.h"
#include "motion.h"
#include "fsm.h"
#include "wdt.h"

const float V_MAX_MPS = 0.5f;
const uint8_t DEAD_DUTY = 40;                          // ASSUMED: measure yours

// ---- the shared state between interrupts and loop(): every one is volatile; multi-byt
volatile uint32_t g_ms = 0;                            // our own millisecond count, fro
volatile uint8_t g_control_due = 0, g_echo_due = 0, g_report_due = 0;
volatile uint16_t g_control_overruns = 0;              // times the control tick came ro
volatile int32_t g_ticks_l = 0, g_ticks_r = 0;

ISR(TIMER1_COMPA_vect) {                               // every 1 ms: keep this SHORT; i
  static uint8_t c20 = 0, c60 = 0;
  static uint16_t c500 = 0;
  ++g_ms;
  if (++c20 >= 20) { c20 = 0; if (g_control_due) ++g_control_overruns; g_control_due = 1
  if (++c60 >= 60) { c60 = 0; g_echo_due = 1; }
  if (++c500 >= 500) { c500 = 0; g_report_due = 1; }
}
ISR(INT0_vect) { bool a = (PIND & _BV(PD2)) != 0, b = (PINC & _BV(PC1)) != 0; g_ticks_l +
ISR(INT1_vect) { bool a = (PIND & _BV(PD3)) != 0, b = (PINC & _BV(PC2)) != 0; g_ticks_r +

static uint32_t now_ms() { uint32_t t; ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { t = g_ms; } r

static void timer1_begin_1khz() {                      // CTC mode, prescaler 1, OCR1A =
  TCCR1A = 0;
  TCCR1B = _BV(WGM12) | _BV(CS10);
  OCR1A = 15999;
  TCNT1 = 0;
  TIMSK1 = _BV(OCIE1A);
}
```

```
static void drive(float vl, float vr) {
  digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
  digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
  analogWrite(PIN_ML_PWM, speed_to_pwm_dz(vl >= 0.0f ? vl : -vl, V_MAX_MPS, DEAD_DUTY));
  analogWrite(PIN_MR_PWM, speed_to_pwm_dz(vr >= 0.0f ? vr : -vr, V_MAX_MPS, DEAD_DUTY));
}

static AvoidConfig CFG = {0.3f, 0.25f, 0.2f, 2.0f, 0.15f, 400, 400, 1200, 3, true};
static TableAvoider machine(CFG, 12345);
static Median<int16_t, 5> median;
static Echo echo;
static Heartbeats<2> beats;                          // 0: control task, 1: echo task
static bool trig_level = false;
static float g_range = -1.0f;

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT); pinMode(PIN_ML_PWM, OUTPUT);
  pinMode(PIN_TRIG, OUTPUT); pinMode(PIN_ECHO, INPUT); pinMode(PIN_BUMP, INPUT_PULLUP);
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP); pinMode(PIN_ENC_L_
  EICRA = _BV(ISC00) | _BV(ISC10); EIFR = _BV(INTF0) | _BV(INTF1); EIMSK = _BV(INT0) | _
  timer1_begin_1khz();
  wdt_enable(WDTO_250MS);                            // from here on, a loop() that st
}

void loop() {
  bool before = trig_level;
  Echo::Status st = echo.update(micros(), digitalRead(PIN_ECHO), trig_level);
  if (trig_level != before) digitalWrite(PIN_TRIG, trig_level);
  if (st == Echo::READY) g_range = (float)median.update((int16_t)(echo_us_to_m(echo.widt
  else if (st == Echo::TIMEOUT) g_range = -1.0f;

  if (g_echo_due) { g_echo_due = 0; echo.start(); beats.checkin(1); }
  if (g_control_due) {
    g_control_due = 0;
    Sense s; s.front_m = g_range; s.right_m = -1.0f; s.bump = digitalRead(PIN_BUMP) == L
    Cmd c = machine.update(s, now_ms());
    drive(c.vl, c.vr);
    beats.checkin(0);
  }
  if (g_report_due) {
    g_report_due = 0;
    int32_t l, r; ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { l = g_ticks_l; r = g_ticks_r; }
    Serial.print(now_ms()); Serial.print(F(" ms  ticks ")); Serial.print(l); Serial.prin
    Serial.print(F("  overruns ")); Serial.println(g_control_overruns);
```

```
    }
    if (beats.all_alive()) { wdt_reset(); beats.clear(); }       // fed only when BOTH tas
  }
```

It compiles for the Uno under `--warnings all` with no warning from the sketch to **6918 bytes of flash (21 %) and 327 bytes of RAM (15 %)**: smaller than Programming for Robotics with Arduino C++ 7's sketch (7114 bytes and 381), since it has no `Scheduler` object, and not comparable with the earlier chapters' sketches, which did different jobs. The disassembly of the Timer1 interrupt shows the price of an ISR in instructions, with no time claimed:

```
Timer1 compare-match interrupt (__vector_11) in robot_ch8.ino, from avr-objdump
  instructions in the whole routine ........ 71   (198 bytes of flash)
  instructions on the usual path ........... 50   (no flag raised: the 20 ms, 60 ms and
    of which: save 7 registers and the status register, then restore them ... 20
    increment of the 32-bit g_ms (4 loads, 3 adds, 4 stores) ................. 11
    three counter tests (20, 60, 500) ....................................... 19
  time taken: NOT measured (no board)
```

Twenty of the 50 instructions are overhead the compiler adds to *every* ISR: seven registers and the status register saved and restored; the useful work is the 32-bit increment of `g_ms` (11 instructions: four loads, three adds and four stores because the counter is 32 bits wide) and three counter checks. A faster ISR would use a 16-bit counter or count in `loop()` from `micros()`. How long those 50 instructions take on a real chip, and how often another interrupt delays this one, **I did not measure**; at 16 MHz an instruction takes between one and four cycles depending on its kind, so the instruction count is an indication of where to look and not a duration.

9. What the numbers can and cannot be trusted for

What they show. The compiler's own output for the real chip: that without `volatile` the wait loop becomes a jump to itself, that a 32-bit read is four loads and an atomic read is eight instructions, the flash and RAM of the sketch, and the number of instructions in the ISR. The calculator's numbers are arithmetic. The shapes from the models: polling misses short pulses by about `w/T`; a cooperative scheduler's worst lateness is the longest other task, if the test covers every phase; a watchdog fed from a timer interrupt catches nothing.

What they do not show. No interrupt latency, execution time or reset was measured; the task times in the scheduler experiment are chosen; the watchdog is a model; the pulse model is Poisson-like with a width I picked; the cost of an RTOS is a description, not a measurement. How the Uno's own bootloader behaves after a watchdog reset is untested.

CHALLENGE 1: WHICH FREQUENCIES CAN A TIMER MAKE?

For every whole frequency from 1 Hz to 20 000 Hz, use the calculator to find the best prescaler and compare value on Timer1 and on Timer2, and report how many frequencies each can reach at all, within 1000 ppm and within 100 ppm, the lowest it can reach and its worst error. Say why the two timers differ.

View solution

CHALLENGE 2: HOW BIG A PIECE?

A 200-character dump at 9600 baud is sent in pieces by a task every 41 ms next to the 20 ms control task. For pieces of 1, 2, 3, 5, 10, 20, 50 and 200 characters, report the control task's worst lateness, the time the dump takes and the share of processor time it uses. Choose the largest piece that keeps the lateness under 2.5 ms, and say what you give up.

View solution

CHALLENGE 3: HOW LONG A WATCHDOG TIMEOUT?

With the supervisor policy, find for each watchdog timeout from 15 ms to 8 s whether a healthy robot is reset by a loop() that is blocked for 0, 4, 17, 42 or 208 ms every 497 ms, and how long a real fault (the control task dying) takes to be caught. Find the rule that predicts the shortest safe timeout for a given blocking time.

View solution

QUICK REFERENCE

- **ISR rules.** Short; no `delay()` or `Serial`; count, copy, set a flag; everything shared with `loop()` is `volatile`.

- **volatile.** Without it, the compiler (avr-gcc 7.3) turned `while (!flag)` into `rjmp .-2`, a jump to itself. With it, a load and a test every time round.
- **Atomic.** A 32-bit read is 4 loads; `ATOMIC_BLOCK` adds `cli` and a status-register restore (8 instructions against 5). Single bytes need no guard; 16 bits and up do.
- **Timers.** CTC frequency = 16 MHz / (prescaler × (OCR + 1)). 1 kHz: Timer1 presc 1, OCR 15999; Timer2 presc 64, OCR 249. Timer2 cannot go below 61 Hz. Timer0 runs `millis()` and the PWM on pins 5 and 6; `Servo` uses Timer1.
- **Polling.** Sees about `min(1, w/T)` of pulses shorter than the loop period; use an interrupt when an event can be shorter than one loop.
- **Cooperative scheduling.** Worst lateness of the control task = the longest other task (3.5 ms report: 2.49; 17.4 ms: 16.37; 208 ms: 207.34, at every phase), but a test with the tasks aligned showed none of it. Five 40-character pieces instead of one 200-character dump: 2.49 ms instead of 16.37.
- **Watchdog.** Feeding in `loop()` catches only a total hang; feeding from a timer ISR catches nothing; the heartbeat supervisor caught all three faults (210–270 ms at a 250 ms timeout). Time to reset is the timeout plus up to one heartbeat period.
- **RTOS.** Preemption and a stack per task; costs RAM and locks; not built or measured here.
- **Budget.** `robot_ch8.ino`: 6918 bytes flash (21 %), 327 bytes RAM (15 %); Timer1 ISR 71 instructions (198 bytes), 50 on the common path. Compiled, not run; nothing timed.
- **Not run:** anything on a board: interrupt latency, execution time, a real watchdog reset, an RTOS.

Next: Programming for Robotics with Arduino C++ 9, Planning and Navigation

Programming for Robotics with Arduino C++ 9 (planned; not yet written) gives the robot a map and a route: occupancy grids stored as bit arrays, breadth-first search and A* within a few kilobytes of RAM, the grid held in flash with `PROGMEM`, and pure-pursuit path following, where the memory budget, rather than the algorithm's elegance, decides how big a map the robot can use.

Programming for Robotics with Arduino C++

Course 2 · Chapter 9 · Planning and Navigation

Programming for Robotics with Arduino C++ 7 gave the robot decisions and Programming for Robotics with Arduino C++ 8 the timing to carry them out; this chapter gives it a *route*. The robot has a map, a start and a goal, and must choose a path that avoids the obstacles and then follow it. Planners are usually taught with lists, dictionaries, sets and priority queues that grow as needed. Here none of that exists: every array has its size fixed at compile time, the Uno has 2048 bytes of RAM, and the question that decides the design is not "which algorithm is elegant?" but "which one fits?".

The chapter builds a grid map held in one bit per cell (and, if it never changes, in flash with no RAM at all), a breadth-first search and an A* planner with fixed-size queues, a line-of-sight smoother, and a pure-pursuit path follower. As in the whole course, the numbers come from running the code, and this chapter has more than its share of my own mistakes: a test world with gaps that closed, and a path follower with two bugs that only the tests found.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): every experiment with an output under it and the three challenge solutions, in a simulated 3 × 2 m arena with four boxes. The map, the range sensor's noise and "ghost" echoes, the robot's 1 cm / 1° localisation error and its speeds are **assumptions I chose**, not measurements. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`): the four-variant size-test sketch and `robot_ch9.ino`. The flash and RAM figures are the compiler's; the symbol sizes come from `avr-nm`. **Nothing was attached to a board and no time was measured**: the chapter reports cells expanded and bytes used, not how long a plan takes on a 16 MHz chip.

1. The budget decides the algorithm

A path planner needs three things: a *map* (which cells are blocked), a *frontier* (the cells waiting to be explored) and some *bookkeeping* (how each cell was reached, so the path can be read back). In Python each is a convenient object that grows: a set of blocked `(x, y)` tuples, a `deque` or a `heapq`, a dictionary of parents. On the Uno each one is an array with a size you chose before the program ran, and what you chose is what you get.

IN PYTHON	ON THE UNO IN C++
a `set` of blocked cells, or a list of lists	one **bit** per cell in a byte array: 600 cells are 75 bytes
a dictionary of parents, `came_from[cell]`	2 bits per cell for the direction the cell was entered from: the parent is one step back
`collections.deque`, `heapq`: grow without limit	a ring buffer and a binary heap of fixed size; a planner that fills them *gives up* and says so, it never overruns
the map is read from a file at start-up	the map can stay in **flash** (`PROGMEM`) and be read with `pgm_read_byte`: no RAM at all
`float` costs, any size of grid	byte-sized costs (a path of up to 254 steps), a grid of a few hundred to a thousand cells

2. The grid, in bits

The whole of the chapter's code is in one header. `BitGrid<W, H>` stores one bit per cell row by row; any class with a `blocked(x, y)` method is a map, and a cell outside the grid counts as blocked, so the planner never has to test the edges. `FlashGrid` has the same layout and reads from a table in flash. `inflate()` makes every obstacle one cell bigger in every direction, so that the planner can treat the robot as a point (the robot has a radius, and a path through the middle of a free cell next to an obstacle is a collision). Below are also the planners, the smoother and the path follower of the next sections.

```
// grid.h -- occupancy grids as bit arrays, breadth-first search and A* in a fixed memory
// Plain C++11: no heap, no Arduino calls except pgm_read_byte for a map kept in flash.
#pragma once
#include <stdint.h>
#include <math.h>
#include "fsm.h"                                        // Cmd, arc(), spin()

#if defined(__AVR__) && !defined(GRID_NO_PROGMEM)
#include <avr/pgmspace.h>
#define GRID_PROGMEM PROGMEM
#define GRID_READ8(p) pgm_read_byte(p)
#else
#define GRID_PROGMEM
#define GRID_READ8(p) (*(p))
```

```
#endif

// ---- 1. the grid: one BIT per cell, row by row. A map is anything with blocked(x, y)
template <uint8_t W, uint8_t H>
class BitGrid {
 public:
  static const uint16_t CELLS = (uint16_t)W * H;
  static const uint16_t BYTES = (uint16_t)((CELLS + 7) / 8);
  BitGrid() { clear(); }
  void clear() { for (uint16_t i = 0; i < BYTES; ++i) b_[i] = 0; }
  bool get(uint8_t x, uint8_t y) const { uint16_t i = (uint16_t)y * W + x; return (b_[i
  void set(uint8_t x, uint8_t y, bool v) {
    uint16_t i = (uint16_t)y * W + x;
    if (v) b_[i >> 3] |= (uint8_t)(1u << (i & 7)); else b_[i >> 3] &= (uint8_t)~(1u << (
  }
  bool blocked(int16_t x, int16_t y) const { return x < 0 || y < 0 || x >= W || y >= H |
  const uint8_t* bytes() const { return b_; }
 private:
  uint8_t b_[BYTES];
};

// the same layout, read straight from a table in flash (PROGMEM): no RAM at all for the
template <uint8_t W, uint8_t H>
class FlashGrid {
 public:
  explicit FlashGrid(const uint8_t* table) : p_(table) {}
  bool blocked(int16_t x, int16_t y) const {
    if (x < 0 || y < 0 || x >= W || y >= H) return true;
    uint16_t i = (uint16_t)y * W + (uint16_t)x;
    return (GRID_READ8(p_ + (i >> 3)) >> (i & 7)) & 1;
  }
 private:
  const uint8_t* p_;
};

// grow every obstacle by r cells in every direction (a square neighbourhood), so the pl
template <class Src, uint8_t W, uint8_t H>
inline void inflate(const Src& src, BitGrid<W, H>& dst, uint8_t r) {
  for (uint8_t y = 0; y < H; ++y)
    for (uint8_t x = 0; x < W; ++x) {
      bool b = false;
      for (int8_t dy = -(int8_t)r; dy <= (int8_t)r && !b; ++dy)
        for (int8_t dx = -(int8_t)r; dx <= (int8_t)r && !b; ++dx) if (src.blocked(x + dx
      dst.set(x, y, b);
    }
```

```
}

// is the straight line between two cells free? (a grid walk: every cell the line crosse
template <class Map>
inline bool line_clear(const Map& m, int16_t x0, int16_t y0, int16_t x1, int16_t y1) {
  int16_t dx = x1 > x0 ? x1 - x0 : x0 - x1, dy = y1 > y0 ? y1 - y0 : y0 - y1;
  int16_t sx = x0 < x1 ? 1 : -1, sy = y0 < y1 ? 1 : -1, err = dx - dy;
  for (;;) {
    if (m.blocked(x0, y0)) return false;
    if (x0 == x1 && y0 == y1) return true;
    int16_t e2 = 2 * err;
    bool mx = e2 > -dy, my = e2 < dx;
    if (mx && my && (m.blocked(x0 + sx, y0) || m.blocked(x0, y0 + sy))) return false;
    if (mx) { err -= dy; x0 += sx; }
    if (my) { err += dx; y0 += sy; }
  }
}

// ---- 2. breadth-first search. Memory: visited bits + 2 bits per cell for "which way d
// plan() returns the number of steps, -1 if the goal cannot be reached, -2 if the queue
// the four neighbours as functions, not as a const table: on an AVR a const array still
inline int8_t dx4(uint8_t d) { return d == 0 ? 1 : (d == 2 ? -1 : 0); }
inline int8_t dy4(uint8_t d) { return d == 1 ? 1 : (d == 3 ? -1 : 0); }

template <uint8_t W, uint8_t H, uint16_t Q>
class Bfs {
 public:
  static const uint16_t CELLS = (uint16_t)W * H;
  uint16_t expanded() const { return expanded_; }
  uint16_t max_frontier() const { return maxq_; }

  template <class Map>
  int16_t plan(const Map& m, uint8_t sx, uint8_t sy, uint8_t gx, uint8_t gy, uint16_t* p
    seen_.clear(); expanded_ = 0; maxq_ = 0;
    if (m.blocked(sx, sy) || m.blocked(gx, gy)) return -1;
    uint16_t head = 0, count = 0;
    uint16_t start = (uint16_t)sy * W + sx, goal = (uint16_t)gy * W + gx;
    seen_.set(sx, sy, true);
    q_[0] = start; count = 1; maxq_ = 1;
    bool found = start == goal;
    while (count && !found) {
      uint16_t c = q_[head]; head = (uint16_t)((head + 1) % Q); --count; ++expanded_;
      uint8_t cx = (uint8_t)(c % W), cy = (uint8_t)(c / W);
      for (uint8_t d = 0; d < 4; ++d) {
        int16_t nx = cx + dx4(d), ny = cy + dy4(d);
```

```
        if (m.blocked(nx, ny) || seen_.get((uint8_t)nx, (uint8_t)ny)) continue;
        if (count >= Q) return -2;
        seen_.set((uint8_t)nx, (uint8_t)ny, true);
        set_dir((uint16_t)ny * W + nx, d);
        q_[(head + count) % Q] = (uint16_t)((uint16_t)ny * W + nx); ++count;
        if (count > maxq_) maxq_ = count;
        if ((uint16_t)ny * W + nx == goal) { found = true; break; }
      }
    }
    if (!found) return -1;
    return trace(start, goal, path, path_cap);
  }

 private:
  void set_dir(uint16_t i, uint8_t d) { uint8_t s = (uint8_t)((i & 3) * 2); dir_[i >> 2]
  uint8_t get_dir(uint16_t i) const { return (dir_[i >> 2] >> ((i & 3) * 2)) & 3; }
  int16_t trace(uint16_t start, uint16_t goal, uint16_t* path, uint16_t cap) const {
    uint16_t n = 0, c = goal;
    while (c != start) { ++n; uint8_t d = get_dir(c); c = (uint16_t)((int16_t)c - dx4(d)
    if (n + 1 > cap) return -2;
    c = goal;
    for (int16_t k = (int16_t)n; k >= 0; --k) { path[k] = c; if (k) { uint8_t d = get_di
    return (int16_t)n;
  }
  BitGrid<W, H> seen_;
  uint8_t dir_[(CELLS + 3) / 4];
  uint16_t q_[Q];
  uint16_t expanded_, maxq_;
};

// ---- 3. A*. Memory: closed bits, 2 bits per cell for the arrival direction, ONE BYTE p
// f = 4 g + wq h, where h is the Manhattan distance to the goal: wq = 4 is the ordinary
// plan() returns the steps, -1 no path, -2 heap full (abandoned), -3 a path longer than
template <uint8_t W, uint8_t H, uint16_t HC>
class AStar {
 public:
  static const uint16_t CELLS = (uint16_t)W * H;
  uint16_t expanded() const { return expanded_; }
  uint16_t max_open() const { return maxh_; }

  template <class Map>
  int16_t plan(const Map& m, uint8_t sx, uint8_t sy, uint8_t gx, uint8_t gy, uint16_t* pa
    closed_.clear(); expanded_ = 0; maxh_ = 0; n_ = 0;
    for (uint16_t i = 0; i < CELLS; ++i) g_[i] = 255;
    if (m.blocked(sx, sy) || m.blocked(gx, gy)) return -1;
```

```
    uint16_t start = (uint16_t)sy * W + sx, goal = (uint16_t)gy * W + gx;
    g_[start] = 0;
    push(f_of(0, sx, sy, gx, gy, wq), start);
    while (n_) {
      uint16_t c = pop();
      uint8_t cx = (uint8_t)(c % W), cy = (uint8_t)(c / W);
      if (closed_.get(cx, cy)) continue;                // a stale duplicate: this cell w
      closed_.set(cx, cy, true); ++expanded_;
      if (c == goal) return trace(start, goal, path, path_cap);
      for (uint8_t d = 0; d < 4; ++d) {
        int16_t nx = cx + dx4(d), ny = cy + dy4(d);
        if (m.blocked(nx, ny) || closed_.get((uint8_t)nx, (uint8_t)ny)) continue;
        uint16_t ni = (uint16_t)ny * W + nx;
        if (g_[c] >= 254) return -3;
        uint8_t ng = (uint8_t)(g_[c] + 1);
        if (ng >= g_[ni]) continue;
        g_[ni] = ng; set_dir(ni, d);
        if (n_ >= HC) return -2;
        push(f_of(ng, (uint8_t)nx, (uint8_t)ny, gx, gy, wq), ni);
      }
    }
    return -1;
  }

 private:
  struct E { uint16_t f, idx; };
  static uint16_t f_of(uint8_t g, uint8_t x, uint8_t y, uint8_t gx, uint8_t gy, uint8_t w
    uint16_t h = (uint16_t)((x > gx ? x - gx : gx - x) + (y > gy ? y - gy : gy - y));
    return (uint16_t)(4u * g + (uint16_t)wq * h);
  }
  void push(uint16_t f, uint16_t idx) {
    uint16_t i = n_++; h_[i].f = f; h_[i].idx = idx;
    while (i && h_[(i - 1) / 2].f > h_[i].f) { E t = h_[i]; h_[i] = h_[(i - 1) / 2]; h_[
    if (n_ > maxh_) maxh_ = n_;
  }
  uint16_t pop() {
    uint16_t top = h_[0].idx;
    h_[0] = h_[--n_];
    uint16_t i = 0;
    for (;;) {
      uint16_t l = (uint16_t)(2 * i + 1), r = (uint16_t)(2 * i + 2), s = i;
      if (l < n_ && h_[l].f < h_[s].f) s = l;
      if (r < n_ && h_[r].f < h_[s].f) s = r;
      if (s == i) break;
      E t = h_[i]; h_[i] = h_[s]; h_[s] = t; i = s;
```

```
    }
    return top;
  }
  void set_dir(uint16_t i, uint8_t d) { uint8_t s = (uint8_t)((i & 3) * 2); dir_[i >> 2]
  uint8_t get_dir(uint16_t i) const { return (dir_[i >> 2] >> ((i & 3) * 2)) & 3; }
  int16_t trace(uint16_t start, uint16_t goal, uint16_t* path, uint16_t cap) const {
    uint16_t n = 0, c = goal;
    while (c != start) { ++n; uint8_t d = get_dir(c); c = (uint16_t)((int16_t)c - dx4(d)
    if (n + 1 > cap) return -2;
    c = goal;
    for (int16_t k = (int16_t)n; k >= 0; --k) { path[k] = c; if (k) { uint8_t d = get_di
    return (int16_t)n;
  }
  BitGrid<W, H> closed_;
  uint8_t g_[CELLS];
  uint8_t dir_[(CELLS + 3) / 4];
  E h_[HC];
  uint16_t n_, expanded_, maxh_;
};

// ---- 4. path smoothing by line of sight: keep a waypoint only where the straight line
// in: `cells` (n cell numbers). out: waypoints in the same array (compacted); returns t
template <class Map>
inline uint16_t shortcut(const Map& m, uint16_t* cells, uint16_t n, uint8_t W) {
  if (n < 3) return n;
  uint16_t out = 1, i = 0;
  while (i < n - 1) {
    uint16_t j = (uint16_t)(n - 1);
    while (j > i + 1 && !line_clear(m, (int16_t)(cells[i] % W), (int16_t)(cells[i] / W),
    cells[out++] = cells[j];
    i = j;
  }
  return out;
}

// ---- 5. pure pursuit: aim at the point on the path one lookahead distance ahead, and
struct Pt { float x, y; };
class PurePursuit {
 public:
  PurePursuit(float lookahead_m, float v_mps, float track_m, float goal_tol_m) : L_(look
  bool done() const { return done_; }
  float target_x() const { return tx_; }
  float target_y() const { return ty_; }
  Cmd update(float x, float y, float th, const Pt* path, uint8_t n) {
    Cmd stop = {0.0f, 0.0f};
```

```
    float gdx = path[n - 1].x - x, gdy = path[n - 1].y - y, gd = sqrtf(gdx * gdx + gdy *
    if (gd < tol_) { done_ = true; return stop; }
    while (idx_ + 2 < n && seg_d2(path[idx_ + 1], path[idx_ + 2], x, y) <= seg_d2(path[i
    if (gd <= L_) { tx_ = path[n - 1].x; ty_ = path[n - 1].y; }
    else {
      tx_ = path[idx_ + 1].x; ty_ = path[idx_ + 1].y;
      for (uint8_t k = idx_; k + 1 < n; ++k) {
        float ax = path[k].x - x, ay = path[k].y - y, bx = path[k + 1].x - x, by = path[
        if (bx * bx + by * by <= L_ * L_) continue;
        float dx = bx - ax, dy = by - ay, a = dx * dx + dy * dy, b = 2 * (ax * dx + ay *
        float disc = b * b - 4 * a * c;
        if (disc < 0) continue;
        float t = (-b + sqrtf(disc)) / (2 * a);
        if (t < 0 || t > 1) continue;
        tx_ = path[k].x + t * dx; ty_ = path[k].y + t * dy;
        break;
      }
    }
    float ang = atan2f(ty_ - y, tx_ - x) - th;
    while (ang > 3.14159265f) ang -= 6.2831853f;
    while (ang < -3.14159265f) ang += 6.2831853f;
    if (ang > 1.2f || ang < -1.2f) return spin(ang > 0 ? 1.5f : -1.5f, track_);
    float dx = tx_ - x, dy = ty_ - y, d = sqrtf(dx * dx + dy * dy); if (d < 0.02f) d = 0
    float curv = 2.0f * sinf(ang) / d;
    float v = v_;
    if (gd < 0.3f) { float vs = 0.05f + 0.8f * gd; if (vs < v) v = vs; }
    return arc(v, v * curv, track_);
  }
 private:
  static float dist2(const Pt& p, float x, float y) { return (p.x - x) * (p.x - x) + (p.y
  static float seg_d2(const Pt& a, const Pt& b, float x, float y) {
    float dx = b.x - a.x, dy = b.y - a.y, l2 = dx * dx + dy * dy, t = l2 > 0 ? ((x - a.x
    if (t < 0) t = 0; else if (t > 1) t = 1;
    float qx = a.x + t * dx - x, qy = a.y + t * dy - y;
    return qx * qx + qy * qy;
  }
  float L_, v_, track_, tol_, tx_ = 0, ty_ = 0;
  uint8_t idx_;
  bool done_;
};
```

Building the map from range readings

A map is rarely typed in; it is built from sensor readings, which are wrong in two ways: the noise moves a hit into a neighbouring cell, and a “ghost” echo (a reading that is too short) marks a cell occupied where nothing is. The experiment scans the test world (four boxes, 158 cells occupied counting the 96 of the outer ring that stands for the arena wall) from eight positions with a sensor that turns through 360°, a reading every degree, 5 mm of noise, 2 % dropouts and ghosts at 0, 1 and 3 %, and keeps each cell in one of three ways: **1 bit** (occupied once *any* reading ends there); a **2-bit hit counter** (occupied when it reaches 2); an **8-bit score** (+3 for a hit, −1 each time a ray passes through the cell, occupied while above 0). It is scored by the cells wrong in each direction and then by what matters: does the planner find a path on the map, with and without inflating it by one cell?

```
// world.h -- the chapter's test world (HOST ONLY): the 3 x 2 m arena of the simulator w
#pragma once
#include <math.h>
#include <stdint.h>
#include "grid.h"
#include "sim.h"

struct Box { double x0, y0, x1, y1; };
static const Box BOXES[4] = {
  {0.8, 0.0, 1.0, 1.2},                                  // a wall stub rising from the b
  {1.8, 0.8, 2.0, 2.0},                                  // a wall hanging from the top e
  {1.4, 1.4, 1.7, 1.7},                                  // a block (0.4 m from the stub:
  {2.4, 1.2, 2.7, 1.5},                                  // a block
};
typedef BitGrid<30, 20> World;
static const double CELL = 0.1;

inline bool in_box(double x, double y) { for (const Box& b : BOXES) if (x >= b.x0 && x <=

// the truth grid: a cell is occupied if any box overlaps it, or if it is in the outermos
inline void truth_grid(World& w) {
  w.clear();
  for (uint8_t y = 0; y < 20; ++y) for (uint8_t x = 0; x < 30; ++x) {
    double cx0 = x * CELL, cy0 = y * CELL, cx1 = cx0 + CELL, cy1 = cy0 + CELL;
    for (const Box& b : BOXES) if (b.x0 < cx1 - 1e-9 && b.x1 > cx0 + 1e-9 && b.y0 < cy1
  }
  for (uint8_t x = 0; x < 30; ++x) { w.set(x, 0, true); w.set(x, 19, true); }
  for (uint8_t y = 0; y < 20; ++y) { w.set(0, y, true); w.set(29, y, true); }
}

// distance along a ray from (px, py) at angle `ang` to the nearest box or arena wall
```

```
inline double cast(double px, double py, double ang) {
  double dx = cos(ang), dy = sin(ang), best = 1e9;
  if (dx > 1e-12) best = fmin(best, (3.0 - px) / dx); else if (dx < -1e-12) best = fmin(
  if (dy > 1e-12) best = fmin(best, (2.0 - py) / dy); else if (dy < -1e-12) best = fmin(
  for (const Box& b : BOXES) {
    double t0 = 0, t1 = 1e9;
    double lo[2] = {b.x0, b.y0}, hi[2] = {b.x1, b.y1}, o[2] = {px, py}, d[2] = {dx, dy};
    bool hit = true;
    for (int i = 0; i < 2; ++i) {
      if (fabs(d[i]) < 1e-12) { if (o[i] < lo[i] || o[i] > hi[i]) hit = false; }
      else { double a = (lo[i] - o[i]) / d[i], c = (hi[i] - o[i]) / d[i]; if (a > c) { d
    }
    if (hit && t0 <= t1 && t0 > 0) best = fmin(best, t0);
  }
  return best;
}
```

```
// e_map.cpp -- building an occupancy grid from range readings, with three ways of rememb
// Eight scan positions in free space; at each, a sensor that turns through 360 degrees t
// (a reading that is too short, uniformly between 0.1 m and the true distance) at a chos
//   1 bit      occupied as soon as ONE reading ends in the cell. Never cleared.   (75 by
//   2 bits     a hit counter 0..3 per cell; occupied when it reaches 2.          (150 by
//   8 bits     a score per cell: +3 for a hit, -1 for each free cell a ray passes throug
// Scored against the true grid: cells wrongly occupied and cells wrongly free, mean over
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "world.h"

struct Pose2 { double x, y; };
static const Pose2 SCAN[8] = {{0.4, 0.4}, {0.4, 1.6}, {1.4, 0.5}, {1.2, 1.1}, {1.6, 1.9},

struct Score { double wrong_occ, wrong_free, ok1, hit1, ok0, hit0; };

static Score run(int mode, double ghost, int seeds) {
  Score s = {0, 0, 0, 0, 0, 0};
  World truth; truth_grid(truth);
  for (int seed = 1; seed <= seeds; ++seed) {
    Rng rng((uint32_t)seed * 40503u + 99u);
    int8_t score[600] = {0}; uint8_t hits[600] = {0};
    World bit;
    for (const Pose2& p : SCAN) {
      for (int deg = 0; deg < 360; ++deg) {
```

```
        double ang = deg * SIM_PI / 180.0, d = cast(p.x, p.y, ang);
        if (rng.uniform() < 0.02) continue;                                  // dropout
        double r = d + 0.005 * rng.normal();
        if (rng.uniform() < ghost) r = 0.1 + rng.uniform() * (d - 0.1 > 0 ? d - 0.1 : 0.(
        if (r < 0.05) continue;
        for (double t = 0.0; t < r - 0.05; t += 0.05) {                      // cells th
          int cx = (int)((p.x + t * cos(ang)) / CELL), cy = (int)((p.y + t * sin(ang)) /
          if (cx >= 0 && cx < 30 && cy >= 0 && cy < 20) { int8_t& q = score[cy * 30 + cx
        }
        int hx = (int)((p.x + r * cos(ang)) / CELL), hy = (int)((p.y + r * sin(ang)) / CE
        if (hx >= 0 && hx < 30 && hy >= 0 && hy < 20 && r < d + 0.2) {       // a hit (a
          int i = hy * 30 + hx;
          bit.set((uint8_t)hx, (uint8_t)hy, true);
          if (hits[i] < 3) ++hits[i];
          if (score[i] <= 17) score[i] += 3; else score[i] = 20;
        }
      }
    }
    World map;
    for (uint8_t y = 0; y < 20; ++y) for (uint8_t x = 0; x < 30; ++x) {
      bool occ = mode == 0 ? bit.get(x, y) : mode == 1 ? hits[y * 30 + x] >= 2 : score[y
      map.set(x, y, occ);
      if (occ && !truth.get(x, y)) s.wrong_occ += 1;
      if (!occ && truth.get(x, y)) s.wrong_free += 1;
    }
    for (int infl = 1; infl >= 0; --infl) {
      World inf; inflate(map, inf, (uint8_t)infl);
      static Bfs<30, 20, 600> bfs; uint16_t path[600];
      int16_t n = bfs.plan(inf, 3, 3, 27, 17, path, 600);
      if (n >= 0) {
        bool hit = false;
        for (int k = 0; k <= n; ++k) if (truth.get((uint8_t)(path[k] % 30), (uint8_t)(pat
        (infl ? s.ok1 : s.ok0) += 1; if (hit) (infl ? s.hit1 : s.hit0) += 1;
      }
    }
  }
  s.wrong_occ /= seeds; s.wrong_free /= seeds; s.ok1 = 100.0 * s.ok1 / seeds; s.hit1 = 10
  return s;
}

int main() {
  World truth; truth_grid(truth);
  int occ = 0; for (uint8_t y = 0; y < 20; ++y) for (uint8_t x = 0; x < 30; ++x) occ += t
  std::printf("occupancy grid from 8 scans of 360 rays; the true grid has %d occupied ce
  std::printf("
```

```
  std::printf("  ghost echoes   memory      wrongly occupied   wrongly free   path found
  const char* names[3] = {"1 bit      ", "2-bit count", "8-bit score"};
  const int bytes[3] = {75, 150, 600};
  for (double g : {0.0, 0.01, 0.03}) for (int mode = 0; mode < 3; ++mode) {
    Score s = run(mode, g, 20);
    std::printf("  %10.0f %%   %s %4d B   %16.1f   %12.1f   %13.0f   %31.0f   %13.0f   %
  }
  return 0;
}
```

```
occupancy grid from 8 scans of 360 rays; the true grid has 158 occupied cells of 600 (th
                                                                    planner on the
  ghost echoes   memory      wrongly occupied   wrongly free   path found (%)   path cro
           0 %   1 bit         75 B               67.8            4.4               0
           0 %   2-bit count  150 B               62.8            5.3               0
           0 %   8-bit score  600 B               15.5           35.1             100
           1 %   1 bit         75 B               84.5            4.3               0
           1 %   2-bit count  150 B               63.5            5.3               0
           1 %   8-bit score  600 B               16.1           35.1             100
           3 %   1 bit         75 B              117.0            4.2               0
           3 %   2-bit count  150 B               67.2            5.2               0
           3 %   8-bit score  600 B               15.2           35.3             100
```

The cell counts are not the interesting column: with the world's edges aligned to cell borders, noise puts hits on *both* sides of every edge, and the 1-bit and 2-bit maps have 63 to 117 cells wrongly occupied against 15 for the 8-bit score. The planner column shows what that costs. With the map inflated by one cell (the safe thing to do) the 1-bit and 2-bit maps leave **no path at all**, in every run at every ghost rate: each obstacle has swollen by about a cell, inflation adds another, and the 0.4 m gap the route must go through is closed. Only the 8-bit score keeps it, because a cell that rays pass through often is cleared again. Without inflation the 1-bit map finds a path in all runs at 0 and 1 % ghosts and in only 60 % at 3 %, where phantom cells sit across the corridor. No planned path went through a truly occupied cell on any map; the 8-bit map has 35 cells wrongly *free* (thin obstacles thinned by the passing rays: I did not check which cells), and none of them mattered here. So **a bit is enough to *store* a map and not always enough to *build* one**: 75 bytes against 600 is what a map that can forget costs. A common compromise is to build with scores (or in a reduced area) and keep only the thresholded bit grid for planning.

3. Breadth-first search and A*

Breadth-first search (BFS) explores the cells in rings of equal distance from the start, so the first time it reaches the goal the path is the shortest. It needs a queue of cells waiting to be explored, a bit per cell for "seen" and two bits per cell for how it got there. **A*** explores the cell with the lowest $f = g + h$, where g is the steps taken so far and h a guess of the steps to go (here the Manhattan distance, which never over-estimates on a 4-connected grid, so A* is still optimal). It explores fewer cells but needs more memory: a priority queue (a binary heap), a closed bit and a byte of g per cell, and the arrival direction. In the code, an entry of the heap is 4 bytes, and a cell can be pushed again when a better route to it is found.

The experiment compares them on 300 random 32 × 32 grids per obstacle density, start at one corner and goal at the other:

```
// e_plan.cpp -- breadth-first search against A* on random 32 x 32 grids (host build; se
// 300 grids per obstacle density, start (0, 0), goal (31, 31) (both kept free). Compare
// both are optimal and must agree), cells expanded, and the biggest queue / open list e
#include <cstdio>
#include <cstring>
#include <initializer_list>
#include "sim.h"
#include "grid.h"

typedef BitGrid<32, 32> Map;

static void random_map(Map& m, double density, uint32_t seed) {
  Rng rng(seed * 2654435761u + 17u);
  m.clear();
  for (uint8_t y = 0; y < 32; ++y) for (uint8_t x = 0; x < 32; ++x) if (rng.uniform() < 
  m.set(0, 0, false); m.set(31, 31, false);
}

template <uint16_t Q, uint16_t HC> struct Planners { Bfs<32, 32, Q> bfs; AStar<32, 32, H

static Planners<1024, 1024> P;                          // big enough that capacity neve

int main() {
  std::printf("breadth-first search against A* (Manhattan heuristic), 32 x 32 grid, 4-co
  std::printf("  obstacle density   solvable   same path length   mean steps   BFS expan
  for (double dens : {0.10, 0.20, 0.30, 0.35}) {
    int solv = 0, same = 0; double steps = 0, eb = 0, ea = 0; uint16_t qmax = 0, hmax = 
    for (int seed = 1; seed <= 300; ++seed) {
```

```
      Map m; random_map(m, dens, (uint32_t)seed);
      uint16_t pb[1024], pa[1024];
      int16_t rb = P.bfs.plan(m, 0, 0, 31, 31, pb, 1024), ra = P.as.plan(m, 0, 0, 31, 31,
      if (rb >= 0 && ra >= 0) {
        ++solv; if (rb == ra) ++same; steps += rb; eb += P.bfs.expanded(); ea += P.as.exp
        if (P.bfs.max_frontier() > qmax) qmax = P.bfs.max_frontier();
        if (P.as.max_open() > hmax) hmax = P.as.max_open();
      } else if ((rb >= 0) != (ra >= 0)) std::printf("  DISAGREE on solvability, seed %d
    }
    std::printf("  %15.0f %%   %8d   %16d   %10.1f   %12.1f   %11.1f   %8.2f   %13u   %1
  }

  std::printf("\ncapacity: the same 300 grids at 20 %% density, planners with a SMALL qu
  std::printf("  capacity (entries)   BFS abandoned   A* abandoned\n");
  static Bfs<32, 32, 32> b32; static Bfs<32, 32, 64> b64; static Bfs<32, 32, 128> b128;
  static AStar<32, 32, 32> a32; static AStar<32, 32, 64> a64; static AStar<32, 32, 128> a
  int ab[3] = {0, 0, 0}, aa[3] = {0, 0, 0};
  for (int seed = 1; seed <= 300; ++seed) {
    Map m; random_map(m, 0.20, (uint32_t)seed);
    uint16_t pt[1024];
    if (P.bfs.plan(m, 0, 0, 31, 31, pt, 1024) < 0) continue;           // only grids th
    if (b32.plan(m, 0, 0, 31, 31, pt, 1024) == -2) ++ab[0];
    if (b64.plan(m, 0, 0, 31, 31, pt, 1024) == -2) ++ab[1];
    if (b128.plan(m, 0, 0, 31, 31, pt, 1024) == -2) ++ab[2];
    if (a32.plan(m, 0, 0, 31, 31, pt, 1024) == -2) ++aa[0];
    if (a64.plan(m, 0, 0, 31, 31, pt, 1024) == -2) ++aa[1];
    if (a128.plan(m, 0, 0, 31, 31, pt, 1024) == -2) ++aa[2];
  }
  const int caps[3] = {32, 64, 128};
  for (int i = 0; i < 3; ++i) std::printf("  %18d   %13d   %12d\n", caps[i], ab[i], aa[i
  return 0;
}
```

```
breadth-first search against A* (Manhattan heuristic), 32 x 32 grid, 4-connected, 300 ra
  obstacle density   solvable   same path length   mean steps   BFS expanded   A* expand
              10 %        292               292         62.0          920.0         40
              20 %        248               248         62.0          816.2         31
              30 %        152               152         63.6          682.6         28
              35 %         71                71         67.9          583.5         27

capacity: the same 300 grids at 20 % density, planners with a SMALL queue (BFS) or heap
  capacity (entries)   BFS abandoned   A* abandoned
                  32             121            248
```

```
                64              0             247
               128              0              70
```

First, they agree: on every solvable grid (292, 248, 152 and 71 of 300 as the density rises) BFS and A* find paths of exactly the same length, and agree on which grids have no path. That is the test of the code as much as of the algorithms: two different implementations that must give the same answer. Second, A* expands 0.39 to 0.47 times as many cells as BFS (407 against 920 at 10 % obstacles). Third, and this is the table that decides the design: the *memory* goes the other way. The biggest queue BFS ever needed was 36 to 40 cells (2 bytes each: 80 bytes), while the biggest open list A* needed was 275 entries at 10 % obstacles (4 bytes each: about 1100 bytes). A* explores fewer cells but holds many more of them, because it pushes duplicates, and its heap is the largest array in the planner.

The second table asks what happens when the queue or the heap is too small: the planner stops with an error code, rather than writing past the end of its array. A BFS queue of 32 entries abandoned 121 of the searches; 64 entries abandoned none. A* with a heap of 32 abandoned 248, 64 abandoned 247, and even 128 abandoned 70 of the roughly 248 solvable grids. Choosing `HC` is the programmer's job and the test is to measure it, as above; the safe choice is the largest ever seen plus a margin, and the error code is the net under it.

4. What fits in 2048 bytes

The sizes follow from the arrays. BFS needs the seen bits (`cells/8` bytes), 2 bits per cell (`cells/4`) and 2 bytes per queue entry. A* needs the closed bits, **one byte per cell** for *g*, 2 bits per cell, and 4 bytes per heap entry. The experiment below applies them to the test world (the 30 × 20 grid inflated by one cell, from cell (3, 3) to (27, 17)) and to several grid sizes, with the queue and heap sized from the largest ever needed on 100 random grids at 10 % obstacles plus 25 %, and a budget of 1500 bytes for map, planner and path buffer (chapter 8's sketch already uses 327 of the 2048):

```
// e_budget.cpp -- what do the planners cost in RAM, and what fits on an Uno? (host buil
// Part 1: the chapter's world (30 x 20 cells, four boxes, inflated by one cell), start
// Part 2: for several grid sizes, the RAM the planner and the map need. The queue and h
//   RAM = map + planner + a path buffer of 2 x 2(W+H) bytes. "fits" means at most 1500 
#include <cstdio>
#include <cmath>
#include <initializer_list>
```

```
#include "world.h"

template <uint8_t W, uint8_t H>
struct Stat { uint16_t q, h; };

template <uint8_t W, uint8_t H>
static Stat<W, H> measure() {
  static Bfs<W, H, (uint16_t)(W * H)> bfs;
  static AStar<W, H, (uint16_t)(W * H * 3 / 2 > 4000 ? 4000 : W * H * 3 / 2)> as;
  static uint16_t path[W * H];
  Stat<W, H> s = {0, 0};
  for (int seed = 1; seed <= 100; ++seed) {
    BitGrid<W, H> m; Rng rng((uint32_t)seed * 2654435761u + 17u);
    for (uint8_t y = 0; y < H; ++y) for (uint8_t x = 0; x < W; ++x) if (rng.uniform() < (
    m.set(0, 0, false); m.set(W - 1, H - 1, false);
    int16_t rb = bfs.plan(m, 0, 0, W - 1, H - 1, path, W * H), ra = as.plan(m, 0, 0, W -
    if (rb >= 0 && ra >= 0) { if (bfs.max_frontier() > s.q) s.q = bfs.max_frontier(); if
  }
  return s;
}

template <uint8_t W, uint8_t H>
static void row() {
  Stat<W, H> s = measure<W, H>();
  uint16_t cells = (uint16_t)(W * H), q = (uint16_t)std::ceil(s.q * 1.25), hc = (uint16_t
  uint16_t mapb = (uint16_t)((cells + 7) / 8), pathb = (uint16_t)(2 * 2 * (W + H));
  uint16_t bfs = (uint16_t)(mapb + (cells + 3) / 4 + 2 * q + 4), as = (uint16_t)(mapb + (
  uint16_t tb = (uint16_t)(mapb + bfs + pathb), ta = (uint16_t)(mapb + as + pathb);
  std::printf("  %2d x %-2d   %5u   %5u   %4u / %-4u   %5u B    %5u B   %-4s   %5u B   %
}

int main() {
  World truth; truth_grid(truth); World inf; inflate(truth, inf, 1);
  static Bfs<30, 20, 600> bfs; static AStar<30, 20, 600> as; uint16_t path[600];
  std::printf("the chapter's world, inflated by 1 cell, from cell (3, 3) to (27, 17)\n");
  std::printf("  planner                steps   cells expanded   biggest queue / open li
  int16_t n = bfs.plan(inf, 3, 3, 27, 17, path, 600);
  std::printf("  BFS                    %5d   %14u   %25u\n", n, bfs.expanded(), bfs.max
  for (int wq : {4, 8, 0}) {
    n = as.plan(inf, 3, 3, 27, 17, path, 600, (uint8_t)wq);
    std::printf("  A* (h weight %d/4)       %5d   %14u   %25u\n", wq, n, as.expanded(), a
  }

  std::printf("\nRAM for the planners (bytes), by grid size\n");
  std::printf("  grid        cells   map B   Q / HC        BFS RAM    A* RAM       BFS to
```

```
  row<16, 16>(); row<24, 24>(); row<30, 20>(); row<32, 32>(); row<40, 40>(); row<64, 64>
  std::printf("\nchecks: sizeof(Bfs<32,32,64>) = %u (formula %u);  sizeof(AStar<32,32,30
              (unsigned)sizeof(Bfs<32, 32, 64>), 128u + 256u + 128u + 4u, (unsigned)size
  return 0;
}
```

```
the chapter's world, inflated by 1 cell, from cell (3, 3) to (27, 17)
  planner                 steps   cells expanded   biggest queue / open list
  BFS                        52              281                          12
  A* (h weight 4/4)          52              212                          47
  A* (h weight 8/4)          54              173                          59
  A* (h weight 0/4)          52              283                          14

RAM for the planners (bytes), by grid size
  grid        cells   map B   Q / HC        BFS RAM    A* RAM       BFS total        A*
  16 x 16       256      32     27 / 97       154 B      746 B   fits     906 B   fits
  24 x 24       576      72     35 / 190      290 B     1558 B   fits    1822 B   no
  30 x 20       600      75     33 / 192      295 B     1599 B   fits    1874 B   no
  32 x 32      1024     128     47 / 320      482 B     2694 B   fits    3078 B   no
  40 x 40      1600     200     58 / 473      720 B     4098 B   fits    4618 B   no
  64 x 64      4096     512     90 / 1123    1720 B    10130 B   no     11154 B   no

checks: sizeof(Bfs<32,32,64>) = 516 (formula 516);  sizeof(AStar<32,32,300>) = 2614 (for
```

On the test world BFS expands 281 cells, A* 212, A* with a heuristic weighted 2 (`wq` = 8, so *h* counts double) 173 cells, but with a path of 54 steps instead of 52; and A* without a heuristic expands 283, as BFS does. The table by size is the real message. **BFS fits up to 40 × 40 cells** (720 bytes for the planner) and is out of budget at 64 × 64 (1720). **A* fits only at 16 × 16**: at 24 × 24 it needs 1558 bytes for the planner alone and at 30 × 20 1599, almost all of it the one-byte-per-cell *g* array and the heap. The two closing checks show the formulas are exact: `sizeof` of the real classes (measured on the PC, which pads a class to an even size; the Uno does not) gives 516 and 2614 bytes, the formulas' 516 and 2614.

The same planners were compiled into a sketch for the Uno, to see what the *compiler* says. The first row has no planner (it has `Serial`, which takes 188 bytes of RAM by itself). The two 30 × 20 rows keep the map in flash, so it costs no RAM:

VARIANT	FLASH (BYTES)	RAM (BYTES)	RAM LEFT
no planner (baseline, `Serial` only)	1710	188	1860

BFS 30 × 20, queue of 40, map in flash	2990	697	1351
A* 30 × 20, heap of 192, map in flash	3582	**1987**	**61**
A* 16 × 16, heap of 97, map in RAM	3470	1166	882

The A* on the test world *compiles and links without a warning*, and leaves **61 bytes** for the stack and everything else. The compiler does not know the stack's depth, so it reports a success for a program that will, sooner or later, overwrite its own variables. This is the point of working out the budget first. The planner's own symbol, read with `avr-nm`, is 0x63F = 1599 bytes, exactly the formula's figure for A* on 30 × 20 with a heap of 192:

```
avr-nm --size-sort -S -C b_p2/s_plan.ino.elf (variant 2: A* on 30 x 20, map in flash), RA
008001df 00000004 b timer0_millis
008001e3 00000004 b timer0_overflow_count
00800100 00000012 d vtable for HardwareSerial
008001e7 0000009d b Serial
00800116 000000c8 b path
00800284 0000063f b planner
```

My first build of this sketch said 705, 1995 and 1174 bytes, 8 more in each than the formula, and the formula was exact for the planner and the path buffer. `avr-nm` found the 8 bytes: the four-neighbour offsets, written as `static const int8_t DX4[4], DY4[4]`. **On an AVR a `const` array is not in flash**; it is a variable in RAM that is initialised from flash at start-up, which is why `PROGMEM` exists. I replaced the tables by two tiny functions (`dx4()` and `dy4()`), and the RAM fell to 697, 1987 and 1166, exactly the predicted figures; the flash grew by two bytes. Challenge 1 asks you to predict these before compiling.

5. Smoothing, and following the path

The grid path is a staircase of 52 steps. A robot could follow the cell centres, but it would turn at every corner, and the path is longer than it needs to be. **Line-of-sight smoothing** keeps a waypoint only where the straight line to the next kept waypoint would cross a blocked cell: the 53 cells become 7 waypoints and the path 4.76 m instead of 5.20 m. **Pure pursuit** follows a path by always steering toward the point on it one *lookahead* distance ahead, along the circle that passes through that point: a curvature of 2 sin α / *d*, where α is the angle between the heading and the target and *d* the distance to it, and a turn rate of speed times curvature. A short

lookahead hugs the path and wobbles; a long one is smooth and cuts corners. The experiment drives the planned path at 0.25 m/s on the chapter-3 simulator with an assumed 1 cm and 1° of localisation error, for five lookaheads, on the raw staircase and on the smoothed path:

```
// e_pursuit.cpp -- following a planned path with pure pursuit, for several lookahead di
// World: world.h (30 x 20 cells of 10 cm, four boxes). The path is A* on the TRUE grid
// 0.25 m/s; its pose comes from the simulator PLUS an assumed localisation error (1 cm
// Scored: reached the goal within 60 s (%), time (s), the true distance of the robot fr
// and collisions (the centre closer than 75 mm, the robot's radius, to a box or a wall:
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "world.h"

static double seg_dist(double px, double py, const Pt& a, const Pt& b) {
  double dx = b.x - a.x, dy = b.y - a.y, l2 = dx * dx + dy * dy, t = l2 > 0 ? ((px - a.x
  if (t < 0) t = 0;
  if (t > 1) t = 1;
  double qx = a.x + t * dx, qy = a.y + t * dy;
  return std::hypot(px - qx, py - qy);
}
static double clearance(double x, double y) {            // distance from the centre to
  double best = std::fmin(std::fmin(x, 3.0 - x), std::fmin(y, 2.0 - y));
  for (const Box& b : BOXES) {
    double dx = std::fmax(std::fmax(b.x0 - x, 0.0), x - b.x1), dy = std::fmax(std::fmax(
    best = std::fmin(best, std::hypot(dx, dy));
  }
  return best;
}

struct R { double reached, time, xtrack, clear, crash; };

static R run(float lookahead, bool smooth, int seeds) {
  World truth; truth_grid(truth);
  World inf; inflate(truth, inf, 1);
  static AStar<30, 20, 300> as;
  uint16_t cells[600];
  int16_t n = as.plan(inf, 3, 3, 27, 17, cells, 600);
  if (n < 0) { std::printf("no path\n"); R z = {0, 0, 0, 0, 0}; return z; }
  uint16_t m = (uint16_t)(n + 1);
  if (smooth) m = shortcut(inf, cells, m, 30);
  Pt path[64];
  for (uint16_t i = 0; i < m && i < 64; ++i) { path[i].x = ((cells[i] % 30) + 0.5f) * 0.
  R r = {0, 0, 0, 0, 0}; double sq = 0; long cnt = 0; double clr = 0;
```

```
  for (int seed = 1; seed <= seeds; ++seed) {
    Rng rng((uint32_t)seed * 31337u);
    RobotParams rp; rp.tau = 0.1;
    SimHW w(rp, SensorParams(), Arena(), (uint32_t)seed);
    w.x = path[0].x; w.y = path[0].y; w.th = 0.0;
    PurePursuit pp(lookahead, 0.25f, 0.15f, 0.06f);
    double minclr = 9.0; bool crashed = false;
    int tick = 0;
    for (; tick < 3000 && !pp.done(); ++tick) {
      float ex = (float)(w.x + 0.01 * rng.normal()), ey = (float)(w.y + 0.01 * rng.norma
      Cmd c = pp.update(ex, ey, eth, path, (uint8_t)m);
      w.drive(c.vl, c.vr);
      for (int k = 0; k < 20; ++k) w.step(0.001);
      double d = 1e9; for (uint16_t i = 0; i + 1 < m; ++i) d = std::fmin(d, seg_dist(w.x
      sq += d * d; ++cnt;
      double cl = clearance(w.x, w.y); if (cl < minclr) minclr = cl; if (cl < 0.075) cra
    }
    if (pp.done()) { r.reached += 1; r.time += tick * 0.02; }
    clr += minclr; if (crashed) r.crash += 1;
  }
  r.time = r.reached ? r.time / r.reached : 0; r.reached = 100.0 * r.reached / seeds; r.
  return r;
}

int main() {
  World truth; truth_grid(truth); World inf; inflate(truth, inf, 1);
  static AStar<30, 20, 300> as; uint16_t cells[600];
  int16_t n = as.plan(inf, 3, 3, 27, 17, cells, 600);
  uint16_t m = (uint16_t)(n + 1), ms = shortcut(inf, cells, m, 30);
  double len = 0;
  for (uint16_t i = 0; i + 1 < ms; ++i) len += std::hypot(((cells[i + 1] % 30) - (cells[
  std::printf("A* path on the inflated grid: %d steps (%.2f m along the grid); after lin
  std::printf("pure pursuit at 0.25 m/s, 20 seeds, assumed 1 cm / 1 degree localisation
  std::printf("  path       lookahead (m)   reached (%%)   time (s)   rms distance from
  for (int smooth = 0; smooth < 2; ++smooth) for (float L : {0.10f, 0.20f, 0.30f, 0.50f,
    R r = run(L, smooth != 0, 20);
    std::printf("  %-9s   %13.2f   %11.0f   %8.1f   %27.1f   %36.0f   %26.0f\n", smooth
  }
  return 0;
}
```

```
A* path on the inflated grid: 52 steps (5.20 m along the grid); after line-of-sight smoo
pure pursuit at 0.25 m/s, 20 seeds, assumed 1 cm / 1 degree localisation error
  path       lookahead (m)   reached (%)   time (s)   rms distance from path (mm)   clos
```

```
raw grid      0.10    100    19.6     7.3
raw grid      0.20    100    18.6    17.1
raw grid      0.30    100    18.0    29.6
raw grid      0.50     95    17.0   114.3
raw grid      0.80    100    13.8   172.0
smoothed      0.10    100    19.0     6.9
smoothed      0.20    100    18.3    20.1
smoothed      0.30    100    17.8    38.5
smoothed      0.50    100    15.7   102.7
smoothed      0.80    100    12.1   206.7
```

On both paths the pattern is the same: **a lookahead of 0.10 to 0.30 m follows the path (7 to 39 mm rms) with no collisions, and one of 0.50 m or more cuts the corners and every run collides** (the robot centre closer than its 75 mm radius to an obstacle, the closest approach is 0, the centre inside the obstacle). The lookahead is also the price of smoothness: the staircase path at 0.10 m takes 19.6 s and at 0.30 m 18.0 s. The smoothed path, 9 % shorter, saves less time than that (19.0 s at 0.10 m) because the speed is fixed and the turns are the same; what it saves is waypoints, 7 against 53. The closest approach at 0.30 m is 83 mm, 8 mm over the limit: the usable range ends there, and challenge 3 maps it against speed.

That table is the third version of it. The first run said that every run on the smoothed path collided at every lookahead, even 0.10 m, with an rms distance from the path of 130 mm, which no sensible controller does on a straight segment. The cause was in `PurePursuit`, and it was two bugs in series. First, the code picked "the segment the robot is on" as the one whose end *waypoint* was nearest, which is correct for a dense staircase and wrong for 7 waypoints a metre apart: halfway along a segment the next waypoint is the nearer, so the lookahead started on the following segment and cut the corner. Fixing that (the nearest *segment*) changed nothing visible, because of the second bug: when the robot was more than one lookahead from the start of its segment, the circle test found nothing and the target silently defaulted to the *goal*, so the robot drove at the goal through the obstacles. The fix is in the header above: aim at the goal only when it is inside the circle, otherwise at the far crossing of the circle with the nearest segment, and otherwise at the next waypoint. **What went wrong:** a default that does something drastic, and a test (the dense staircase) that never reached the case. **How to avoid it:** test a follower on a sparse path as well as a dense one, and make a default the *safe* action, not the end of the route.

6. On the Uno

For a map that does not change, there is no need to build the grid on the robot at all. The program `e_gen.cpp` builds the test world on the PC, inflates it by one cell and *writes the map as a C++ table*, 75 bytes in flash:

```
// world_pm.h -- generated by e_gen.cpp: the 30 x 20 world inflated by one cell, one bit
const uint8_t WORLD_PM[75] GRID_PROGMEM = {
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x3F, 0x78, 0x00, 0x00, 0x0F,
  0x1E, 0x00, 0xC0, 0x83, 0x07, 0x00, 0xF0, 0xE0, 0x01, 0x00, 0x3C, 0x78,
  0x00, 0x00, 0x0F, 0x1E, 0x78, 0xC0, 0x83, 0x07, 0x1E, 0xF0, 0xE0, 0x81,
  0x07, 0x3C, 0x78, 0xE0, 0x01, 0x0F, 0x1E, 0x78, 0xFE, 0x83, 0x07, 0x9E,
  0xFF, 0x00, 0xF8, 0xE7, 0x3F, 0x00, 0xFE, 0xF9, 0x0F, 0x80, 0x7F, 0xFE,
  0x03, 0xE0, 0x1F, 0xF0, 0x00, 0xF8, 0x07, 0xFC, 0xFF, 0xFF, 0xFF, 0xFF,
  0xFF, 0xFF, 0xFF,
};
```

`robot_ch9.ino` plans once at start-up with BFS (the 309-byte planner of the table; the biggest queue ever needed on this map was 12 cells, the queue holds 40) over a `FlashGrid`, smooths the route, and then follows it with pure pursuit at a 0.2 m lookahead on a pose from the wheel encoders (the midpoint update of Programming for Robotics with Arduino C++ 5), at 20 ms steps on the scheduler of Programming for Robotics with Arduino C++ 1. The wheels get the speeds as open-loop commands, without the speed controllers of Programming for Robotics with Arduino C++ 6.

```
// robot_ch9.ino -- Programming for Robotics with Arduino C++, chapter 9: plan a route o
// Compile-checked for the Uno (arduino:avr:uno). NOT run on a board: nothing was attach
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include "scheduler.h"
#include "conv.h"
#include "pins.h"
#include "motion.h"
#include "grid.h"
#include "world_pm.h"                                  // WORLD_PM: the 30 x 20 map infl

const float V_MAX_MPS = 0.5f;
const uint8_t DEAD_DUTY = 40;                          // ASSUMED: measure yours
const float WHEEL_R = 0.033f, TRACK = 0.150f;
const float PER_TICK = 6.2831853f * WHEEL_R / 360.0f;
const float CELL_M = 0.1f;
```

```
volatile int32_t g_ticks_l = 0, g_ticks_r = 0;
ISR(INT0_vect) { bool a = (PIND & _BV(PD2)) != 0, b = (PINC & _BV(PC1)) != 0; g_ticks_l -
ISR(INT1_vect) { bool a = (PIND & _BV(PD3)) != 0, b = (PINC & _BV(PC2)) != 0; g_ticks_r -

static void drive(float vl, float vr) {
  digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
  digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
  analogWrite(PIN_ML_PWM, speed_to_pwm_dz(vl >= 0.0f ? vl : -vl, V_MAX_MPS, DEAD_DUTY));
  analogWrite(PIN_MR_PWM, speed_to_pwm_dz(vr >= 0.0f ? vr : -vr, V_MAX_MPS, DEAD_DUTY));
}

static Bfs<30, 20, 40> planner;                       // 309 bytes of RAM (the queue ho
static uint16_t cells[100];                           // the planned route, as cell numb
static Pt waypoints[16];                              // the smoothed route in metres
static uint8_t n_way = 0;
static PurePursuit pursuit(0.20f, 0.25f, TRACK, 0.06f);
static Pose pose = {0.35f, 0.35f, 0.0f};
static int32_t last_l = 0, last_r = 0;
static Scheduler<2> sch;

static void control_task(void*, uint32_t) {
  int32_t l, r; ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { l = g_ticks_l; r = g_ticks_r; }
  float dl = (float)(l - last_l) * PER_TICK, dr = (float)(r - last_r) * PER_TICK;
  last_l = l; last_r = r;
  pose_midpoint(pose, 0.5f * (dl + dr), (dr - dl) / TRACK);
  if (n_way < 2 || pursuit.done()) { drive(0.0f, 0.0f); return; }
  Cmd c = pursuit.update(pose.x, pose.y, pose.th, waypoints, n_way);
  drive(c.vl, c.vr);
}
static void report_task(void*, uint32_t now) {
  Serial.print(now); Serial.print(F(" ms  pose ")); Serial.print(pose.x, 2); Serial.print
  Serial.print(F("  waypoints ")); Serial.print(n_way); Serial.println(pursuit.done() ? "
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT); pinMode(PIN_ML_PWM, OUTPUT);
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP); pinMode(PIN_ENC_L_B
  EICRA = _BV(ISC00) | _BV(ISC10); EIFR = _BV(INTF0) | _BV(INTF1); EIMSK = _BV(INT0) | _B
  FlashGrid<30, 20> map(WORLD_PM);
  int16_t steps = planner.plan(map, 3, 3, 27, 17, cells, 100);                // plan on
  if (steps >= 0) {
    uint16_t m = shortcut(map, cells, (uint16_t)(steps + 1), 30);
    for (uint16_t i = 0; i < m && i < 16; ++i) { waypoints[i].x = ((cells[i] % 30) + 0.5
```

```
  }
  Serial.print(F("plan: ")); Serial.print(steps); Serial.print(F(" steps, ")); Serial.pr
  sch.add(control_task, 0, 20);
  sch.add(report_task, 0, 500);
}

void loop() { sch.run(millis()); }
```

It compiles for the Uno under `--warnings all` with no warning from the sketch to **11 388 bytes of flash (35 %) and 943 bytes of RAM (46 %)**. The flash is the largest of the course so far, because `sinf`, `atan2f` and `sqrtf` of the path follower and the float printing come from the floating-point library; the RAM is the planner (309), the route (200 and 128 bytes) and the rest of the robot. It was *not run*: no time per plan or per control step was measured, and whether BFS and A* choose the same route among equally short ones, so that the smoothed waypoints are the ones of section 5, was not checked.

7. What the numbers can and cannot be trusted for

What they show. The sizes: the formulas for BFS and A* are exact against `sizeof` and the compiler's own symbol sizes, and A* on a 600-cell grid leaves the Uno with 61 bytes. The shapes: A* expands fewer cells and holds more; the two planners agree on every path length; a bit map with a 1-cell inflation can close a corridor that a score map keeps; a lookahead beyond about 0.3 m cuts corners in this world.

What they do not show. The world is one small arena with four boxes whose edges happen to align with the grid, which is the worst case for the bit maps; the random grids are not realistic rooms; the sensor, the ghosts, the localisation error and the 0.25 m/s are assumptions; the pure pursuit used a robot pose with only random noise, not the drift of real odometry (Programming for Robotics with Arduino C++ 5). **No time was measured**, so the claim that A* "is faster" is a claim about cells expanded, not about seconds, and with a heap in the loop it may not hold on a chip that cannot afford the heap.

CHALLENGE 1: PREDICT THE RAM

From the sizes of the arrays in `grid.h`, predict the "Global variables" figure the compiler will report for each of the four variants of the size-test sketch (baseline of 188 bytes, a path buffer of 200 bytes, the 30

× 20 map in flash, a 16 × 16 map in RAM), then compare with `sizeof` and with the compiler's report. Say how much stack the A* variants leave.

View solution

CHALLENGE 2: A WEIGHTED HEURISTIC

Run A* with the heuristic weighted by 0, 1, 1.25, 1.5, 2, 3 and 4 on 300 random 32 × 32 grids at 20 % obstacles. Report the cells expanded, the longest open list, the mean path, the share of grids on which the path is longer than the optimal one, and the worst excess. Choose a weight for a robot that cannot afford a heap of 177 entries, and say what you give up.

View solution

CHALLENGE 3: LOOKAHEAD AGAINST SPEED

Drive the planned path at 0.15, 0.25, 0.35 and 0.50 m/s with lookaheads of 0.10, 0.20, 0.30 and 0.40 m, 20 runs each, and report the share of runs with a collision and the time to the goal. Find the largest safe lookahead at each speed and say what happens to the time as the lookahead grows.

View solution

QUICK REFERENCE

- **Grid.** One bit per cell (600 cells = 75 bytes); outside counts as blocked; a map in flash with `PROGMEM` costs no RAM; inflate by the robot's radius so the planner can treat it as a point.
- **Building a map.** 1-bit and 2-bit maps spill across edges (63–117 wrong cells against 15 for an 8-bit score) and, inflated by 1 cell, closed the only corridor in every run; the 8-bit score kept it in every run.
- **BFS against A*.** Same path length on every grid; A* expands 0.39–0.47 of the cells (407 against 920 at 10 %) but its heap needed 275 entries (about 1100 bytes) against 40 queue cells (80 bytes).
- **Capacity.** A planner whose queue or heap is full returns an error, never overruns: BFS queue 32 abandoned 121 searches, 64 none; A* heap 128 abandoned 70.
- **RAM formulas.** BFS = cells/8 + cells/4 + 2Q + 4. A* = cells/8 + cells + cells/4 + 4HC + 6. Exact against `sizeof` (516, 2614) and `avr-nm` (1599 for A* 30 × 20, heap 192).
- **Fits in 1500 bytes.** BFS up to 40 × 40; A* only at 16 × 16. A* 30 × 20 compiles and leaves 61 bytes of stack.

- **const is RAM on an AVR.** Two 4-byte `const` tables cost 8 bytes of RAM; two inline functions cost none (RAM 705→697).
- **Smoothing and pure pursuit.** 53 cells → 7 waypoints, 5.20 m → 4.76 m. Lookahead 0.10–0.30 m: 7–39 mm rms, no collisions; 0.50 m and up: collisions in every run. Target the goal only when it is inside the circle; test on a sparse path.
- **Budget.** `robot_ch9.ino`: 11 388 bytes flash (35 %), 943 bytes RAM (46 %), compiled, not run; no time measured.
- **Not run:** anything on a board; time per plan or per step; real odometry drift; any map built from a real sensor.

Next: Programming for Robotics with Arduino C++ 10, State Estimation

Programming for Robotics with Arduino C++ 10 (planned; not yet written) is about knowing where the robot is: the complementary filter, a one-dimensional Kalman filter and a two-by-two position-and-speed filter with the matrices written out, fusing odometry with an IMU, and the count of operations that makes float against fixed-point the central question on a chip with no floating-point unit.

Programming for Robotics with Arduino C++

Course 2 · Chapter 10 · State Estimation

Programming for Robotics with Arduino C++ 9 gave the robot a route; to follow it the robot must know where it is, and no sensor says so. The wheel encoders drift (Programming for Robotics with Arduino C++ 5), a gyro integrates its own offset into a heading that wanders off, a compass is noisy and is thrown by every motor and steel table, an accelerometer shakes with the motors. Each is right about something the others are wrong about. *State estimation* is the craft of combining them: believing each sensor where it is good and ignoring it where it is bad.

The chapter builds three tools in order of weight: the **complementary filter** (one line), a **scalar Kalman filter** (one number and a gain) and a **two-state Kalman filter** with its small matrices multiplied out into ordinary arithmetic. Then it fuses a gyro with a compass, and wheel speed with a range sensor, and ends with the question the Uno forces: what does each of them cost on a chip with no floating-point unit, and when is integer arithmetic worth the trouble?

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): every experiment with an output under it and the three challenge solutions, on *simulated* sensors. The gyro offsets and noise, the compass noise and its 20° disturbances, the accelerometer vibration, the range noise and dropouts, and the odometry noise are **assumptions I chose**, not measurements of any real part. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`): the six-variant cost-test sketch and `robot_ch10.ino`; the instruction and library-call counts come from `avr-objdump`, the sizes from the compiler and `avr-nm`. **From memory and not verified:** the gyro chip's I2C address, its register numbers and its scale in `robot_ch10.ino` are what I remember of an MPU-6050-style data sheet; I did not check them and the code never ran against a chip. **Nothing was attached to a board and no time was measured.** "Cost" here means bytes of flash and RAM and the number of instructions and library calls: never seconds.

1. Why one sensor is not enough

The three failures have different shapes, and the filters of this chapter work because the shapes are different. A gyro measures how fast the robot turns, so its angle is the sum of many readings: smooth and quick to respond, but any constant offset in the readings becomes an angle that grows without limit. A compass or an accelerometer measures the angle itself, so it

never drifts, but it is noisy and, near motors, disturbed. A good estimate follows the gyro for the fast changes and the absolute sensor for the slow ones. In Python this is a few lines of NumPy; on the Uno it is a few lines of arithmetic whose price you can read off the compiler's report.

IN PYTHON	ON THE UNO IN C++
`F @ P @ F.T + Q` on NumPy arrays, any size of state	the 2 × 2 products multiplied out by hand into scalar statements: 13 multiplications for a predict and an update
`np.linalg.inv(S)` for the innovation covariance	a measurement of one number makes `S` a scalar: one division
64-bit floats at the speed of the hardware	32-bit floats done by library routines: every `+`, `*`, `/` is a function call of a couple of hundred bytes of code
`filterpy` or a library fixes the maths for you	nothing: you choose `Q` and `R`, and they are assumptions about hardware you must measure
no thought for the word size	integers in micro-radians, gains in Q16: you choose the units so nothing overflows and nothing rounds to zero

2. The complementary filter

The complementary filter is the cheapest estimator there is: `angle = alpha * (angle + rate * dt) + (1 - alpha) * measured`. Each step the angle moves with the gyro (the part in the brackets) and is then pulled a little way toward the absolute measurement. The weight `alpha = tau / (tau + dt)` is set by one number, the **time constant** `tau`: for changes faster than about `tau` seconds the gyro is believed, for slower ones the measurement. All the estimators of the chapter are in one header, `est.h`: plain C++11, no heap, no matrix library, no Arduino calls, so the same file runs on the PC and the Uno. Here it is, in four parts (the complementary filter and its fixed-point twin, a scalar Kalman filter, a two-state Kalman filter, and a fixed-point heading filter):

```
// est.h -- state estimation in small, fixed-size pieces (Programming for Robotics with A
// The two-state filters are written out as scalar equations: on a chip with no floating
#pragma once
#include <stdint.h>

// ---- 1. complementary filter: trust the gyro for fast changes and the accelerometer (
// angle = alpha * (angle + rate dt) + (1 - alpha) * measured,  alpha = tau / (tau + dt)
```

```
class Complementary {
 public:
  Complementary(float tau_s, float dt_s, float start = 0.0f) : angle_(start), alpha_(tau
  float update(float rate, float measured, float dt_s) { angle_ = alpha_ * (angle_ + rat
  float angle() const { return angle_; }
 private:
  float angle_, alpha_;
};

// the same filter in integers: angle in micro-radians, rate in micro-radians per second
class ComplementaryFixed {
 public:
  ComplementaryFixed(float tau_s, int32_t start_urad = 0) : angle_(start_urad), k_((int3
  int32_t update(int32_t rate_urad_s, int32_t measured_urad) {
    angle_ += rate_urad_s / 100;                                  // 10 ms
    int32_t d = measured_urad - angle_;
    if (d > 2000000) d = 2000000; else if (d < -2000000) d = -2000000;      // keeps the
    angle_ += ((d >> 4) * k_) >> 12;
    return angle_;
  }
  int32_t k() const { return k_; }
 private:
  int32_t angle_, k_;
};

// ---- 2. a scalar Kalman filter: one number that changes slowly (a distance, a bias), r
class Kalman1 {
 public:
  Kalman1(float x0, float p0, float q, float r) : x_(x0), p_(p0), q_(q), r_(r), k_(0.0f)
  void update(float z) {
    p_ += q_;                                                   // predict: the state
    k_ = p_ / (p_ + r_);                                        // the gain: how much
    x_ += k_ * (z - x_);
    p_ *= (1.0f - k_);
  }
  float x() const { return x_; }
  float p() const { return p_; }
  float gain() const { return k_; }
 private:
  float x_, p_, q_, r_, k_;
};

// ---- 3. a two-state Kalman filter with the matrices written out. State (x0, x1); meas
// model:   x0' = x0 + f01 * x1 * dt + u * dt        x1' = x1          (f01 = +1: x0 is
// F = [1  f01 dt; 0  1],   Q = diag(q0 dt, q1 dt) (a simple choice, not the textbook ma
```

```
class Kalman2 {
 public:
  Kalman2(float f01, float q0, float q1, float p0 = 1.0f, float p1 = 1.0f) : f01_(f01), (
  void set(float x0, float x1) { x0_ = x0; x1_ = x1; }
  void predict(float u, float dt) {
    x0_ += (f01_ * x1_ + u) * dt;                                  // x = F x + B u
    float a = p00_ + f01_ * dt * p01_;                             // P = F P F^T + Q, m
    float b = p01_ + f01_ * dt * p11_;
    p00_ = a + f01_ * dt * b + q0_ * dt;                           //   (symmetric: p10 =
    p01_ = b;
    p11_ += q1_ * dt;
  }
  void update_x0(float z, float r) {                               // H = [1 0]
    float s = p00_ + r;                                            // innovation varianc
    k0_ = p00_ / s; k1_ = p01_ / s;                                // the gain K = P H^T
    float y = z - x0_;
    x0_ += k0_ * y; x1_ += k1_ * y;
    float p00 = p00_, p01 = p01_;                                  // P = (I - K H) P
    p00_ -= k0_ * p00; p01_ -= k0_ * p01; p11_ -= k1_ * p01;
  }
  void update_x1(float z, float r) {                               // H = [0 1]
    float s = p11_ + r;
    k0_ = p01_ / s; k1_ = p11_ / s;
    float y = z - x1_;
    x0_ += k0_ * y; x1_ += k1_ * y;
    float p01 = p01_, p11 = p11_;
    p00_ -= k0_ * p01; p01_ -= k0_ * p11; p11_ -= k1_ * p11;
  }
  float x0() const { return x0_; }
  float x1() const { return x1_; }
  float k0() const { return k0_; }
  float k1() const { return k1_; }
  float p00() const { return p00_; }
 private:
  float f01_, q0_, q1_, x0_, x1_, p00_, p01_, p11_, k0_, k1_;
};

// ---- 4. the same heading-and-bias filter in integers, with the gains fixed (they sett
// heading in micro-radians, bias in micro-radians per second, gyro rate in micro-radian
class HeadingFixed {
 public:
  HeadingFixed(int32_t k0_q16, int32_t k1_q16) : th_(0), b_(0), k0_(k0_q16), k1_(k1_q16)
  void set(int32_t th_urad, int32_t b_urad_s) { th_ = th_urad; b_ = b_urad_s; }
  void predict(int32_t rate_urad_s) { th_ += (rate_urad_s - b_) / 100; }
  void update(int32_t z_urad) {
```

```
    int32_t y = z_urad - th_;
    if (y > 2000000) y = 2000000; else if (y < -2000000) y = -2000000;
    th_ += ((y >> 4) * k0_) >> 12;
    b_ += ((y >> 4) * k1_) >> 12;
  }
  int32_t heading() const { return th_; }
  int32_t bias() const { return b_; }
 private:
  int32_t th_, b_, k0_, k1_;
};
```

The first experiment is a simulated tilt: two sine waves and a step of 0.4 rad at 30 s (the robot is tipped up a kerb). The gyro has a constant offset (random per run, up to 0.03 rad/s) and noise; the accelerometer angle has noise and a 6 Hz vibration from the motors. All of that is assumed. The table is the rms error in degrees, over the whole run and over the 5 s after the step (how fast the filter follows a real change):

```
// e_comp.cpp -- the complementary filter on a simulated tilt angle (host build; every s
// Truth: the robot's tilt (rad) is two sine waves (periods 8 s and 2.5 s) and a step of
//   gyro            the true rate + a constant offset (random per run, +-0.03 rad/s) + r
//   accelerometer   the true angle + noise of 0.05 rad + a vibration of 0.12 rad at 6 H
// Compared: the gyro integrated alone, the accelerometer angle alone, and the complemen
// Scored: the rms error over the run (degrees) and over the 5 s after the step (how qui
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "est.h"

static double truth(double t) { return 0.35 * std::sin(2 * SIM_PI * t / 8.0) + 0.12 * st
static double truth_rate(double t) { return 0.35 * 2 * SIM_PI / 8.0 * std::cos(2 * SIM_P

struct Err { double all, after; };

// kind: -1 gyro only, -2 accelerometer only, otherwise the filter's tau in seconds
static Err run(double tau, int kind, double vib_hz, int seeds) {
  double sq_all = 0, sq_after = 0; long n_all = 0, n_after = 0;
  for (int seed = 1; seed <= seeds; ++seed) {
    Rng rng((uint32_t)seed * 7777u + 5u);
    double bias = (rng.uniform() * 2 - 1) * 0.03;
    const float dt = 0.01f;
    Complementary f(kind > 0 ? (float)tau : 1.0f, dt, (float)truth(0));
```

```
      double gyro_int = truth(0);
      for (int i = 1; i <= 6000; ++i) {
        double t = i * dt;
        double rate = truth_rate(t) + bias + 0.01 * rng.normal();
        double acc = truth(t) + 0.05 * rng.normal() + 0.12 * std::sin(2 * SIM_PI * vib_hz *
        gyro_int += rate * dt;
        double est = kind == -1 ? gyro_int : kind == -2 ? acc : f.update((float)rate, (flo
        double e = est - truth(t);
        sq_all += e * e; ++n_all;
        if (t > 30.0 && t <= 35.0) { sq_after += e * e; ++n_after; }
      }
    }
    Err r = {std::sqrt(sq_all / n_all) * 57.2958, std::sqrt(sq_after / n_after) * 57.2958};
    return r;
  }

  int main() {
    std::printf("tilt estimate: rms error in degrees, mean over 40 runs (vibration 6 Hz)\n
    std::printf("  estimator                    whole run   5 s after the step\n");
    { Err r = run(0, -1, 6.0, 40); std::printf("  gyro integrated alone       %10.2f   %18
    { Err r = run(0, -2, 6.0, 40); std::printf("  accelerometer angle alone   %10.2f   %18
    for (double tau : {0.05, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0}) { Err r = run(tau, 1, 6.0, 40
    return 0;
  }
```

```
tilt estimate: rms error in degrees, mean over 40 runs (vibration 6 Hz)
  estimator                    whole run   5 s after the step
  gyro integrated alone            37.52               40.13
  accelerometer angle alone         5.65                5.66
  complementary, tau  0.05 s        2.35                2.60
  complementary, tau  0.20 s        1.22                3.12
  complementary, tau  0.50 s        1.59                5.02
  complementary, tau  1.00 s        2.30                7.25
  complementary, tau  2.00 s        3.51               10.47
  complementary, tau  5.00 s        6.53               16.22
  complementary, tau 10.00 s       10.76               21.20
```

Neither sensor is good alone: the gyro's offset drifts the angle by 37° rms in a minute, the accelerometer is 5.7° off all the time. Between them there is a best `tau`, here 0.2 s, at 1.22°: a quarter of the accelerometer's error. But the best `tau` is a *trade*: a long one trusts the gyro longer, so its offset has time to drift the angle (3.5° at 2 s) and a real step is followed slowly

(10.5° in the 5 s after it); a short one lets the vibration through. The best value depends on what is wrong with the absolute sensor, which is challenge 1.

3. A Kalman filter with one state

The complementary filter has one fixed weight. A Kalman filter *computes* the weight, the **gain** `K`, from two numbers you give it: how noisy a measurement is (`R`) and how much the state may change between measurements (`q`). It keeps a third number `P`, how uncertain it currently is, which grows by `q` each step and shrinks when a measurement arrives. Start knowing nothing and the first measurement is nearly all believed (`K` near 1); as confidence grows, `K` falls. Let me see how the gain behaves on a range sensor with 5 mm of noise, one reading every 60 ms (assumed), for several values of `q`:

```
// e_kalman1.cpp -- the scalar Kalman filter: what the gain does (host build; the sensor
// Part 1: the gain K step by step for a range that does not change (truth 1.000 m), for
//         The filter starts knowing nothing (variance 1 m^2). Steady state from the for
// Part 2: the same filters following a range that moves at 0.05 m/s (3 mm per reading).
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "est.h"

int main() {
  const double R = 0.005 * 0.005;
  const double qs[4] = {0.0, 1e-8, 1e-6, 1e-4};
  std::printf("part 1: the gain K after k readings of a fixed range (sigma 5 mm, R = %.1
  std::printf("  k     q = 0      q = 1e-8   q = 1e-6   q = 1e-4\n");
  Kalman1 f[4] = {Kalman1(1.0f, 1.0f, (float)qs[0], (float)R), Kalman1(1.0f, 1.0f, (floa
  Rng rng(12345);
  int shown[] = {1, 2, 3, 4, 5, 10, 20, 50, 100, 1000};
  int si = 0;
  for (int k = 1; k <= 1000; ++k) {
    double z = 1.0 + 0.005 * rng.normal();
    for (int i = 0; i < 4; ++i) f[i].update((float)z);
    if (k == shown[si]) { std::printf("  %-4d  %.5f    %.5f    %.5f    %.5f\n", k, f[0].
  }
  std::printf("  formula, steady state:");
  for (int i = 0; i < 4; ++i) { double pm = (qs[i] + std::sqrt(qs[i] * qs[i] + 4 * qs[i]
  std::printf("\n  (q = 0: K = 1/k exactly after k readings, a running average that neve
```

```
  std::printf("\npart 2: following a range that moves at 0.05 m/s, readings every 60 ms f
  std::printf("  q (m^2 per reading)   steady gain K   rms error (mm)   rms error of the
  for (double q : {0.0, 1e-8, 1e-6, 1e-5, 1e-4}) {
    double sq = 0, sqraw = 0; long n = 0;
    for (int seed = 1; seed <= 200; ++seed) {
      Rng r2((uint32_t)seed * 911u);
      Kalman1 k(1.0f, 1.0f, (float)q, (float)R);
      for (int i = 1; i <= 333; ++i) {
        double truth = 1.0 + 0.05 * i * 0.06;
        double z = truth + 0.005 * r2.normal();
        k.update((float)z);
        if (i > 10) { sq += (k.x() - truth) * (k.x() - truth); sqraw += (z - truth) * (z
      }
    }
    Kalman1 g(1.0f, 1.0f, (float)q, (float)R); for (int i = 0; i < 2000; ++i) g.update(1
    std::printf("  %19.0e   %13.5f   %14.2f   %34.2f\n", q, g.gain(), std::sqrt(sq / n) *
  }
  return 0;
}
```

```
part 1: the gain K after k readings of a fixed range (sigma 5 mm, R = 2.5e-05 m^2)
  k     q = 0      q = 1e-8   q = 1e-6   q = 1e-4
  1     0.99997    0.99997    0.99997    0.99997
  2     0.50034    0.50044    0.51013    0.83337
  3     0.33348    0.33371    0.35489    0.82857
  4     0.25008    0.25043    0.28310    0.82843
  5     0.20005    0.20053    0.24420    0.82843
  10    0.10001    0.10115    0.18855    0.82843
  20    0.05000    0.05245    0.18113    0.82843
  50    0.02000    0.02606    0.18100    0.82843
  100   0.01000    0.02055    0.18100    0.82843
  1000  0.00100    0.01980    0.18100    0.82843
  formula, steady state:  0.00000  0.01980  0.18100  0.82843
  (q = 0: K = 1/k exactly after k readings, a running average that never stops averaging

part 2: following a range that moves at 0.05 m/s, readings every 60 ms for 20 s, 200 runs
  q (m^2 per reading)   steady gain K   rms error (mm)   rms error of the raw readings (m
               0e+00         0.00050           292.15                                  4
               1e-08         0.01980           126.17                                  4
               1e-06         0.18100            13.61                                  4
               1e-05         0.46332             4.43                                  4
               1e-04         0.82843             4.25                                  4
```

With `q = 0` the gain is exactly 1/k after k readings: the filter is a running average that never stops averaging, and when the range moves it follows at a rms error of 292 mm, *far worse than the raw readings' 5 mm*. The gain settles at a value set only by `q` and `R` (the steady states match the formula to the digits shown), and the choice of `q` is the whole tuning: 1e-8 is still 126 mm off, 1e-6 is 13.6 mm, and from 1e-5 the filter is better than the raw sensor (4.4 and 4.25 mm against 4.99). A filter told that nothing moves stops listening; a Kalman filter is only as good as the model you tell it.

4. Two states, with the matrices written out

Most of what a robot needs has two numbers tied together: a position and its speed, or a heading and the gyro's offset. The state is `x = (x0, x1)`; between steps `x0` advances by `f01 * x1 * dt` (plus a known input `u`, the gyro rate, for the heading) and `x1` stays. In matrices:

```
predict:   x = F x + B u                  F = [ 1   f01*dt ]      f01 = +1: x0 is a po
           P = F P F^T + Q                    [ 0     1    ]      f01 = -1: x0 is
                                                                   (the gyro reads

update:    s = P00 + R                    (measuring x0: H = [1 0])
           K = P H^T / s = (P00, P01) / s   y = z - x0
           x = x + K y
           P = (I - K H) P

multiplied out for the 2 x 2 symmetric P = [p00 p01; p01 p11]:
           a = p00 + f01 dt p01    b = p01 + f01 dt p11
           p00 = a + f01 dt b + q0 dt     p01 = b     p11 = p11 + q1 dt
```

There is one measurement at a time, so `s` is a single number and the "inverse" is one division; the only matrix algebra left is the 2 × 2 products, which `Kalman2` in `est.h` has written out so you can count the operations. The process noise here is the simple `Q = diag(q0 dt, q1 dt)`, not the textbook matrix derived from a continuous model: a choice made for clarity, not the only correct one. `update_x1` measures the second state instead (the speed from the wheels), with `H = [0 1]`.

Heading from a gyro and a compass

Here `x0` is the heading and `x1` the gyro's offset: the gyro is not a measurement but the *input* `u` of the predict step at 100 Hz, and the compass (every 100 ms) is the measurement. The filter

has no direct reading of the offset, yet it learns it: the gyro's heading and the compass keep disagreeing in the same direction, and the filter puts that disagreement down to `x1`. The experiment compares six estimators over 120 s, with the robot turning back and forth, and the compass disturbed by +20° for 5 s at 60 s and again at 100 s (assumed: a motor or a steel table). The fifth estimator ignores a compass reading more than 0.27 rad (three of its standard deviations) from the estimate: a **gate**. The sixth is the integer `HeadingFixed`, with the Kalman gains frozen at their settled values, computed on the PC:

```
// e_heading.cpp -- heading from a gyro and a compass: six ways (host build; every senso
// Truth: the robot sits still for 2 s, then turns back and forth at 0.8 sin(2 pi t / 20
//   gyro      the true rate + an offset (random per run, +-0.02 rad/s = +-1.1 deg/s) th
//   compass   the true heading + noise of 0.09 rad (5 deg), and a magnetic DISTURBANCE:
// Estimators: (1) the gyro alone, its offset measured while the robot sits still for th
// (4) the two-state Kalman filter (heading, gyro offset) with the compass as measuremen
// (6) the fixed-point filter with the Kalman gains of (4) frozen at their settled value
#include <cstdio>
#include <cmath>
#include "sim.h"
#include "est.h"

struct Out { double rms, rms_dist, worst, bias_err; };

static double rate_at(double t) { return t < 2.0 ? 0.0 : 0.8 * std::sin(2 * SIM_PI * (t

// settled gains of the Kalman filter at the moments of a compass update (the predict ru
static void settled_gains(double q0, double q1, double r, float& k0, float& k1) {
  Kalman2 k(-1.0f, (float)q0, (float)q1, 1.0f, 0.01f);
  for (int i = 0; i < 400; ++i) { for (int j = 0; j < 10; ++j) k.predict(0.0f, 0.01f); k
  k0 = k.k0(); k1 = k.k1();
}

static Out run(int kind, int seeds, float k0q = 0, float k1q = 0) {
  const double q0 = 0.005 * 0.005 * 0.01, q1 = 0.0005 * 0.0005, r = 0.09 * 0.09;
  double sq = 0, sqd = 0, worst = 0, berr = 0; long n = 0, nd = 0;
  for (int seed = 1; seed <= seeds; ++seed) {
    Rng rng((uint32_t)seed * 4271u + 3u);
    double bias = (rng.uniform() * 2 - 1) * 0.02, th = 0.0;
    double gyro_th = 0.0, bias_cal = 0.0; int ncal = 0;
    Complementary comp(2.0f, 0.01f, 0.0f);
    Kalman2 kf(-1.0f, (float)q0, (float)q1, 1.0f, 0.01f), kg(-1.0f, (float)q0, (float)q1
    HeadingFixed fx((int32_t)std::lround(k0q * 65536.0), (int32_t)std::lround(k1q * 6553
    double compass = 0.0;
    for (int i = 1; i <= 12000; ++i) {
```

```
      double t = i * 0.01;
      double rate = rate_at(t);
      th += rate * 0.01;
      bias += 0.0005 * std::sqrt(0.01) * rng.normal();
      double gyro = rate + bias + 0.005 * rng.normal();
      if (t <= 2.0) { bias_cal += gyro; ++ncal; }
      gyro_th += (gyro - (ncal ? bias_cal / ncal : 0.0)) * 0.01;
      bool dist = (t >= 60 && t < 65) || (t >= 100 && t < 105);
      bool have = i % 10 == 0;
      if (have) compass = th + 0.09 * rng.normal() + (dist ? 0.35 : 0.0);
      double est = 0, bias_est = 0;
      comp.update((float)gyro, (float)compass, 0.01f);
      kf.predict((float)gyro, 0.01f); kg.predict((float)gyro, 0.01f);
      fx.predict((int32_t)std::lround(gyro * 1e6));
      if (have) {
        kf.update_x0((float)compass, (float)r);
        if (std::fabs(compass - kg.x0()) < 0.27) kg.update_x0((float)compass, (float)r);
        fx.update((int32_t)std::lround(compass * 1e6));
      }
      switch (kind) {
        case 1: est = gyro_th; break;
        case 2: est = compass; break;
        case 3: est = comp.angle(); break;
        case 4: est = kf.x0(); bias_est = kf.x1(); break;
        case 5: est = kg.x0(); bias_est = kg.x1(); break;
        default: est = fx.heading() * 1e-6; bias_est = fx.bias() * 1e-6; break;
      }
      double e = est - th;
      sq += e * e; ++n; if (std::fabs(e) > worst) worst = std::fabs(e);
      if (dist) { sqd += e * e; ++nd; }
      if (i == 12000) berr += std::fabs(bias_est - bias);
    }
  }
  Out o = {std::sqrt(sq / n) * 57.2958, std::sqrt(sqd / nd) * 57.2958, worst * 57.2958, I
  return o;
}

int main() {
  float k0, k1;
  settled_gains(0.005 * 0.005 * 0.01, 0.0005 * 0.0005, 0.09 * 0.09, k0, k1);
  std::printf("heading from a gyro (100 Hz) and a compass (10 Hz, two 5 s disturbances o
  std::printf("settled Kalman gains at a compass update: K0 = %.5f (heading), K1 = %.6f
  std::printf("  estimator                                  rms error (deg)   rms in the
  const char* names[6] = {"gyro alone, offset calibrated at start", "compass alone", "co
  for (int kind = 1; kind <= 6; ++kind) {
```

```
    Out o = run(kind, 100, k0, k1);
    if (kind == 1 || kind == 2 || kind == 3) std::printf("  %-40s   %15.2f   %29.2f   %1
    else std::printf("  %-40s   %15.2f   %29.2f   %17.2f   %31.3f\n", names[kind - 1], o
  }
  return 0;
}
```

```
heading from a gyro (100 Hz) and a compass (10 Hz, two 5 s disturbances of 20 degrees),
settled Kalman gains at a compass update: K0 = 0.01866 (heading), K1 = -0.001743 (gyro o
  estimator                                  rms error (deg)   rms in the disturbances (
  gyro alone, offset calibrated at start               9.39                          1
  compass alone                                        7.90                          2
  complementary filter, tau 2 s                        5.02                          1
  Kalman, heading + gyro offset                        4.41                          
  Kalman with a gate at 0.27 rad                       1.84                          
  fixed-point, gains frozen (Q16)                      4.46                          
```

The ranking is not what the first intuition says. Each sensor alone is poor (9.4° and 7.9° rms; the compass is 20.5° off during the disturbances). The complementary filter helps (5.0°), and the Kalman filter is better again (4.4°) because it also tracks the gyro's offset: a 0.33°/s error in it at the end of the run. But the two disturbances dominate everything: with the *gate*, which costs one comparison, the same filter is at 1.84° rms and 3.1° in the disturbances, 0.096°/s from the true offset. The expensive part of a filter was not the matrices; it was deciding when to ignore a sensor. The gate has a cost of its own that this run does not show: if the filter's estimate were ever far wrong, it could keep rejecting the compass that would correct it (I did not test that case). And the integer filter with frozen gains is 4.46° against the float's 4.41°: the arithmetic is not what limits the answer.

Position and speed along a wall

The same filter with `f01 = +1` estimates a position and its speed. A range sensor gives the position every 60 ms (5 mm noise, 2 % of the readings missing); the wheels give the speed every 20 ms (8 mm/s noise). The robot accelerates, cruises and brakes. Compared: the raw reading and its difference over 60 ms; the filter with the range only; the filter with the range *and* the odometry speed, fused through `update_x1`:

```
// e_posvel.cpp -- position and speed along a wall from a range sensor and wheel odometry
// Truth: the robot starts at 300 mm from the wall's reference, speeds up to 200 mm/s ov
//   range      every 60 ms: the true position + noise of 5 mm; 2 % of the readings are
```

```
//   odometry   every 20 ms: the true speed + noise of 8 mm/s
// Estimators: (1) the range reading itself, and its difference over 60 ms as the speed;
// (3) the same filter also fed with the odometry speed (update_x1) every 20 ms. Units: m
// Scored: rms position error and rms speed error over the run, and over the 2 s of brak
#include <cstdio>
#include <cmath>
#include "sim.h"
#include "est.h"

static double speed_at(double t) {
  if (t < 2.0) return 100.0 * t;
  if (t < 8.0) return 200.0;
  if (t < 10.0) return 200.0 - 100.0 * (t - 8.0);
  return 0.0;
}

struct Out { double pos, spd, pos_b, spd_b; };

static Out run(int kind, int seeds, double q1) {
  double sp = 0, ss = 0, spb = 0, ssb = 0; long n = 0, nb = 0;
  for (int seed = 1; seed <= seeds; ++seed) {
    Rng rng((uint32_t)seed * 911u + 17u);
    double x = 300.0, z = 300.0, zprev = 300.0, vz = 0.0;
    Kalman2 kf(1.0f, 0.01f, (float)q1, 100.0f, 10000.0f);
    kf.set(300.0f, 0.0f);
    for (int i = 1; i <= 600; ++i) {                    // 20 ms steps, 12 s
      double t = i * 0.02, v = speed_at(t);
      x += v * 0.02;
      kf.predict(0.0f, 0.02f);
      if (i % 3 == 0) {
        if (rng.uniform() >= 0.02) {
          z = x + 5.0 * rng.normal();
          vz = (z - zprev) / 0.06; zprev = z;
          if (kind >= 2) kf.update_x0((float)z, 25.0f);
        } else {
          zprev = z;                                    // missing reading: hold the las
          vz = 0.0;
        }
      }
      if (kind == 3) kf.update_x1((float)(v + 8.0 * rng.normal()), 64.0f);
      double ep, ev;
      if (kind == 1) { ep = z - x; ev = vz - v; } else { ep = kf.x0() - x; ev = kf.x1()
      sp += ep * ep; ss += ev * ev; ++n;
      if (t > 8.0 && t <= 10.0) { spb += ep * ep; ssb += ev * ev; ++nb; }
    }
```

```
  }
  Out o = {std::sqrt(sp / n), std::sqrt(ss / n), std::sqrt(spb / nb), std::sqrt(ssb / nb
  return o;
}

int main() {
  std::printf("position and speed along a wall, 12 s, mean over 200 runs (position in mm
  std::printf("  estimator                                    position rms   speed rms   p
  const char* names[3] = {"range reading, differenced for speed", "Kalman, range only",
  for (int kind = 1; kind <= 3; ++kind) {
    Out o = run(kind, 200, 2000.0);
    std::printf("  %-40s   %12.2f   %9.2f   %22.2f   %19.2f\n", names[kind - 1], o.pos,
  }
  return 0;
}
```

```
position and speed along a wall, 12 s, mean over 200 runs (position in mm, speed in mm/s
  estimator                                  position rms   speed rms   position rms (br
  range reading, differenced for speed               6.65      121.06
  Kalman, range only                                 3.44       17.03
  Kalman, range + odometry speed                     1.52        4.94
```

Differentiating a noisy reading is the worst way to get a speed: 121 mm/s rms against a top speed of 200 (the raw position is 6.65 mm rms, worse than the sensor's 5 mm, because a missing reading repeats the last one). The filter with the range alone gets the speed to 17 mm/s and the position to 3.4 mm; the odometry speed, whose own noise (8 mm/s) is larger than the filter's result, still improves both, to 4.9 mm/s and 1.5 mm, because the two sensors' errors are independent. Here they are independent by construction: real wheels slip exactly when the range and the odometry disagree, and a real fusion needs a gate on the odometry too.

5. What it costs on the Uno

Everything above ran on a PC with hardware floating point. On the Uno every `float` operation is a call to a library routine, and every 32-bit integer division or multiplication is as well. The size test `s_est.ino` has six variants, each with one estimator step in a function of its own, fed from `analogRead` and printed as an integer so the compiler can neither remove it nor add the float printing code to some variants and not others. Variant 0 is the baseline with no estimator.

```
// s_est.ino -- cost test for chapter 10: the estimators on the Uno. Compile-checked only
// VARIANT 0: baseline (nothing), 1: complementary float, 2: complementary fixed-point, 
// Each estimator step is a separate noinline function so avr-objdump can count its instr
#include "est.h"

#ifndef VARIANT
#define VARIANT 1
#endif

#if VARIANT == 1
static Complementary est(0.5f, 0.01f);
__attribute__((noinline)) float step(float rate, float meas) { return est.update(rate, m
#elif VARIANT == 2
static ComplementaryFixed est(0.5f);
__attribute__((noinline)) int32_t step_fx(int32_t rate, int32_t meas) { return est.updat
#elif VARIANT == 3
static Kalman1 est(0.0f, 1.0f, 1e-6f, 25.0f);
__attribute__((noinline)) float step(float, float meas) { est.update(meas); return est.x
#elif VARIANT == 4
static Kalman2 est(-1.0f, 2.5e-7f, 2.5e-7f);
__attribute__((noinline)) float step(float rate, float meas) { est.predict(rate, 0.01f);
#elif VARIANT == 5
static HeadingFixed est(1223, -114);
__attribute__((noinline)) int32_t step_fx(int32_t rate, int32_t meas) { est.predict(rate
#endif

void setup() { Serial.begin(115200); }

void loop() {
  int a = analogRead(A0), b = analogRead(A1);      // inputs the compiler cannot know
#if VARIANT == 0
  Serial.println((int32_t)(a + b) * 1000L);
#elif VARIANT == 2 || VARIANT == 5
  Serial.println(step_fx((int32_t)(a - 512) * 1000L, (int32_t)(b - 512) * 1000L));
#else
  Serial.println((int32_t)(step((a - 512) * 0.01f, (b - 512) * 0.001f) * 1000.0f));   //
#endif
  delay(10);
}
```

Compiled for the Uno with `--warnings all` (no warning from the sketch; the Arduino core's own `new.cpp` printed four on the first build, before the core was cached):

```
  variant                              flash (bytes)   vs baseline   RAM (bytes)   vs base
  0  baseline (nothing)                     1960             -           188
  1  complementary, float                   2994         +1034           196          +
  2  complementary, fixed-point             2336          +376           196          +
  3  Kalman1, float                         3348         +1388           208         +2
  4  Kalman2 predict + update, float        3926         +1966           228         +4
  5  HeadingFixed predict + update          2460          +500           204         +1
```

Then the step function of each variant, from `avr-objdump` (script `count_est.py`): how many instructions it has and which library routines it calls. Calls are counted where they are written, not where they run; there are no loops in these functions, so each call is made once per step:

```
  complementary, float              154 bytes   71 instructions   calls: __addsf3 x2, __
  complementary, fixed-point        214 bytes   93 instructions   calls: __divmodsi4 x1,
  Kalman1, float                    278 bytes  111 instructions   calls: __addsf3 x3, __
  Kalman2 predict + update, float   778 bytes  304 instructions   calls: __addsf3 x10, _
  HeadingFixed predict + update     314 bytes  130 instructions   calls: __divmodsi4 x1,
```

And the size of the routines those calls reach, from `avr-nm`:

```
  $ avr-nm -S b_e4/s_est.ino.elf | grep the library routines: symbol sizes in bytes (stati
  __addsf3x   204 bytes
  __divsf3      8 bytes
  __divsf3x   220 bytes
  __mulsf3      8 bytes
  __mulsf3x   210 bytes
  __subsf3     14 bytes
  __divmodsi4   46 bytes
  __mulsi3     32 bytes
```

Three readings. **The first float operation is the expensive one.** The complementary filter's step is 154 bytes, but the sketch grew by 1034, because the routines for add, subtract and multiply (about 200 bytes each) come with it. Programming for Robotics with Arduino C++ 9's sketch already carried them, so there the marginal cost of a float filter is closer to the step function itself: 154 bytes for the complementary filter, 778 for `Kalman2`. **The two-state filter is about five times the code of the one-line one** (778 bytes, 304 instructions, 29 library calls against 154 bytes, 71 instructions, 6 calls), and the 2 × 2 matrices are the reason. **Fixed-point is not free of library calls either.** `ComplementaryFixed` has an instruction count *higher* than the float one (93 against 71) because of the division by 100 in `rate / 100`, which is a call to a 32-bit divide, and the multiplication by the gain is a call to a 32-bit multiply. On an AVR the integer

routines are far smaller (46 and 32 bytes against about 200 for each float one) and I expect them to be faster; *I did not time anything*, so that is an expectation and not a result. What I can say from the table is that `HeadingFixed`, doing the work of the 304-instruction float filter with frozen gains, is 130 instructions, 314 bytes and 3 calls, and a quarter of the flash cost when the float library is not otherwise needed (+500 against +1966).

What fixed-point asks of you in exchange is design work the float version never needed. The units must be chosen so that a value fits: micro-radians keep the heading to 1 µrad in a 32-bit integer, and a gain is stored as an integer in Q16. Here `y` is clamped to ±2 000 000 and shifted by 4 before the multiplication, because `y * gain` in a plain `int32_t` overflows. The gains must be frozen: the full filter's gains change while it converges, and a fixed gain is only right once they have settled, so the first seconds are worse (challenge 3 shows how many bits the gains need).

6. The robot

`robot_ch10.ino` is the heading filter on the robot. A gyro on the I2C bus is read every 10 ms and used as the input of the predict step, the wheel-encoder heading (the sum of the wheel differences as in Programming for Robotics with Arduino C++ 5) corrects it every 100 ms, the scheduler of Programming for Robotics with Arduino C++ 1 runs the three tasks, and the position is advanced along the estimated heading. At start-up it averages 200 readings with the robot still to take off most of the gyro's offset, so the filter only has to learn the remainder. If the gyro stops answering, a few missed readings reuse the last rate and many mean the input is zero and the report says `GYRO LOST`: a filter that is silently fed a stale input is worse than one that knows it has none.

```
// robot_ch10.ino -- Programming for Robotics with Arduino C++, chapter 10: the heading
// Compile-checked for the Uno (arduino:avr:uno). NOT run on a board: nothing was attach
// its register numbers (0x6B power management, 0x47 gyro Z high byte) and its scale (13
// NOT verified against it and NOT run. Check them before you trust a number from this s
#include <Wire.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include "scheduler.h"
#include "pins.h"
#include "motion.h"
#include "est.h"
```

```
const uint8_t GYRO_ADDR = 0x68;
const float GYRO_LSB_PER_RAD_S = 131.0f * 57.29578f;           // 131 LSB per degree/sec
const float WHEEL_R = 0.033f, TRACK = 0.150f;
const float PER_TICK = 6.2831853f * WHEEL_R / 360.0f;

volatile int32_t g_ticks_l = 0, g_ticks_r = 0;
ISR(INT0_vect) { bool a = (PIND & _BV(PD2)) != 0, b = (PINC & _BV(PC1)) != 0; g_ticks_l +
ISR(INT1_vect) { bool a = (PIND & _BV(PD3)) != 0, b = (PINC & _BV(PC2)) != 0; g_ticks_r +

static bool gyro_read(int16_t& raw) {                           // one gyro Z reading; fa
  Wire.beginTransmission(GYRO_ADDR); Wire.write(0x47);
  if (Wire.endTransmission(false) != 0) return false;
  if (Wire.requestFrom(GYRO_ADDR, (uint8_t)2) != 2) return false;
  raw = (int16_t)((Wire.read() << 8) | Wire.read());
  return true;
}

static float gyro_offset = 0.0f;                                // rad/s, measured at sta
static Kalman2 heading(-1.0f, 2.5e-7f, 2.5e-7f, 1.0f, 0.01f);   // x0 = heading (rad), x1
static float enc_heading = 0.0f;                                // the heading from the w
static Pose pose = {0.0f, 0.0f, 0.0f};
static int32_t last_l = 0, last_r = 0;
static float last_rate = 0.0f;
static bool gyro_ok = false;
static uint8_t misses = 0;
static Scheduler<3> sch;

static void predict_task(void*, uint32_t) {                     // every 10 ms: predict w
  int16_t raw;
  if (gyro_read(raw)) { last_rate = (float)raw / GYRO_LSB_PER_RAD_S - gyro_offset; misses
  else if (misses < 255) { ++misses; if (misses > 5) gyro_ok = false; }   // a few misses
  heading.predict(gyro_ok ? last_rate : 0.0f, 0.01f);
  int32_t l, r; ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { l = g_ticks_l; r = g_ticks_r; }
  float dl = (float)(l - last_l) * PER_TICK, dr = (float)(r - last_r) * PER_TICK;
  last_l = l; last_r = r;
  enc_heading += (dr - dl) / TRACK;
  float m = heading.x0();
  pose.x += 0.5f * (dl + dr) * cosf(m); pose.y += 0.5f * (dl + dr) * sinf(m); pose.th = m
}
static void update_task(void*, uint32_t) {                      // every 100 ms: correct
  heading.update_x0(enc_heading, 0.01f);
}
static void report_task(void*, uint32_t now) {
  Serial.print(now); Serial.print(F(" ms  heading ")); Serial.print(heading.x0() * 57.295
  Serial.print(F(" deg  gyro offset ")); Serial.print(heading.x1() * 57.29578f, 2);
```

```
  Serial.print(F(" deg/s  wheels ")); Serial.print(enc_heading * 57.29578f, 1); Serial.p
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP); pinMode(PIN_ENC_L_
  EICRA = _BV(ISC00) | _BV(ISC10); EIFR = _BV(INTF0) | _BV(INTF1); EIMSK = _BV(INT0) | _
  Wire.begin();
  Wire.beginTransmission(GYRO_ADDR); Wire.write(0x6B); Wire.write((uint8_t)0); Wire.endT
  float sum = 0.0f; uint8_t n = 0;
  for (uint8_t i = 0; i < 200; ++i) { int16_t raw; if (gyro_read(raw)) { sum += (float)r
  if (n > 100) { gyro_offset = sum / n; gyro_ok = true; }
  Serial.print(F("gyro offset ")); Serial.print(gyro_offset * 57.29578f, 3); Serial.prin
  sch.add(predict_task, 0, 10);
  sch.add(update_task, 0, 100);
  sch.add(report_task, 0, 500);
}

void loop() { sch.run(millis()); }
```

It compiles for the Uno under `--warnings all` with no warning from the sketch to:

```
Sketch uses 9726 bytes (30%) of program storage space. Maximum is 32256 bytes.
Global variables use 574 bytes (28%) of dynamic memory, leaving 1474 bytes for local var
```

That is 9 726 bytes of flash and 574 bytes of RAM. It was **not run**: no chip was attached, the I2C address, register numbers and scale are from my memory of a data sheet and unverified, and the filter's `Q` (2.5e-7 for both states) is a guess that must be tuned on the real gyro. Wheels that slip make the corrections wrong in exactly the situation they are needed, which is why a real robot would gate them as in section 4.

7. What the numbers can and cannot be trusted for

What they show. The shapes: a complementary filter has a best time constant that depends on the disturbance; a Kalman filter's gain settles at a value computed from `q` and `R` (matching the formula), and a wrong `q` can be worse than no filter; a filter that estimates the gyro's offset beats one that does not; a gate on the measurement matters more than the arithmetic; a speed from differencing is hopeless and one from fusion is not; the counts of instructions and library calls of each step, and the flash and RAM of each variant on the Uno.

What they do not show. Every sensor is a simulation with noise I chose: real gyro noise is not Gaussian and white, real offsets depend on temperature, a real compass is not merely off by 20° for 5 s, and real vibration is not one sine wave. The filter's `Q` and `R` were set from the simulation's own parameters, which a real robot does not know; and the filters are optimal *only* when the model matches, which here it does by construction. **No time was measured**: the Uno figures count bytes, instructions and library calls; a claim of speed (integer against float) would need a timer on a board. `robot_ch10.ino` was compiled, not run.

CHALLENGE 1: THE BEST TIME CONSTANT AGAINST THE VIBRATION

Repeat the tilt experiment with the accelerometer's vibration at 0.5, 2, 5 and 20 Hz, and for each find the best time constant of the complementary filter among 0.05, 0.1, 0.2, 0.5, 1, 2 and 5 s. Say what the best `tau` does as the vibration gets faster, and why.

View solution

CHALLENGE 2: TUNE THE POSITION AND SPEED FILTER

Sweep the speed process noise `q1` (10 to 100 000) and the range noise `R` the filter is told (4, 25 and 250, when the true value is 25) for the range-only position and speed filter. Report the rms position and speed error and the *lag*: the time after the robot has stopped for the speed estimate to fall below 10 mm/s. What do `q1` and `R` trade off, and what happens if you tell the filter the wrong `R`?

View solution

CHALLENGE 3: BITS FOR THE GAINS

The fixed-point heading filter stores its gains as integers with `F` fractional bits. For `F` = 4 to 20, report the stored gains, the rms heading error and the error of the offset estimate, and compare with the float filter. How many bits are needed, and which gain is the one that decides it?

View solution

QUICK REFERENCE

- **Complementary filter.** `angle = alpha (angle + rate dt) + (1 - alpha) measured`, `alpha = tau / (tau + dt)`. Tilt: gyro alone 37.5° rms, accelerometer alone 5.65°, best `tau` 0.2 s at 1.22°; 0.2 s was

best at 6 Hz vibration, 1 s at 0.5 Hz and 0.1 s at 20 Hz.

- **Scalar Kalman.** `P += q; K = P / (P + R); x += K (z - x); P *= 1 - K`. With `q = 0`, `K = 1/k`: a filter told nothing moves is 292 mm off a moving range; `q` 1e-5 and up beats the raw 4.99 mm (4.4 mm).
- **Two states.** `F = [1 f01 dt; 0 1]`, `Q = diag(q0 dt, q1 dt)` (a simple choice), one measurement at a time so `s` is a scalar. Heading + gyro offset: `f01 = -1`, gyro as input. Position + speed: `f01 = +1`.
- **Heading, 120 s, with two 20° compass disturbances.** gyro 9.39°, compass 7.90°, complementary 5.02°, Kalman 4.41°, Kalman with a 0.27 rad gate 1.84° (3.14° inside the disturbances), fixed-point with frozen gains 4.46°. A gate is worth more than the arithmetic.
- **Position and speed.** Differencing the range: 121 mm/s rms. Kalman, range only: 3.44 mm and 17.0 mm/s. With the odometry speed: 1.52 mm and 4.94 mm/s.
- **Cost on the Uno (compiled, not run).** Flash over a 1960-byte baseline: complementary float +1034, fixed +376, Kalman1 +1388, Kalman2 +1966, HeadingFixed +500. Steps: 71, 93, 111, 304, 130 instructions; Kalman2 makes 29 float library calls, HeadingFixed 3 integer ones. Integer division by 100 is a library call too.
- **Fixed-point rules.** Use micro-radians, not a per-step rounding of tiny increments; clamp the input before the multiply; freeze gains only after they settle; a gain that rounds to zero stops the filter learning.
- **Budget.** `robot_ch10.ino`: 9 726 bytes flash (30 %), 574 bytes RAM (28 %), compiled, not run; the gyro's register map is from memory and unverified; no time measured.
- **Not run:** anything on a board; any real gyro, compass or range sensor; the speed of float against integer; the filters' `Q` and `R` on real hardware.

Next: Programming for Robotics with Arduino C++ 11, Robot Arms in Code

Programming for Robotics with Arduino C++ 11 (planned; not yet written) moves from a robot that drives to one that reaches: forward and inverse kinematics of a two-link arm, trapezoidal speed profiles, the limits of the Servo library, pick-and-place on the simulator, and lookup tables in flash for the trigonometry the chip cannot afford to compute.

Programming for Robotics with Arduino C++

Course 2 · Chapter 11 · Robot Arms in Code

Everything so far has been a robot that drives. This chapter is about one that *reaches*: a small arm of two links and three hobby servos that picks something up at one point and puts it down at another. The mathematics is old and short (a triangle and the law of cosines); what makes it a chapter for the Uno is what surrounds it: the servos take whole degrees or microseconds, the library that drives them takes a timer and a pin you may have promised to something else, the trigonometry is slow on a chip with no floating-point unit, and an arm that moves its joints together does not move its tip in a straight line.

The chapter builds the arm as plain functions (forward and inverse kinematics), a trapezoidal speed profile, a set of lookup tables in flash, and a pick-and-place cycle that is first run on the simulator and then written as a sketch for the Uno. As in the rest of the course, every number comes from running the code, and what was not run is said so. The arm itself is an assumption: **no arm was built, and the Uno never drove a servo**.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): every experiment with an output under it and the three challenge solutions, on a *simulated* arm: links of 100 and 80 mm, a table at −40 mm, joint limits of 120°/s and 400°/s², and 0.8° of play in each joint are **assumptions I chose**, not measurements. **Read from source:** what the Servo library does (its pulse range, its integer-degree `write`, its 20 ms refresh, the timer it takes) comes from the source of Servo 1.3.0 that I installed with `arduino-cli` for this chapter; I read it, I did not run it on a board. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`): the eight-variant cost-test sketch and `robot_ch11.ino`; instruction and library-call counts come from `avr-objdump`, sizes from the compiler and `avr-nm`. **Nothing was attached to a board and no time was measured.** "Cost" means bytes of flash and RAM and the number of instructions and library calls, never seconds; the claims about real servos (speed, jitter, torque, how a pulse becomes an angle) are not tested here.

1. An arm is a function from angles to a point

A planar arm with two links is described by three numbers: the lengths `l1` and `l2` and, for a given pose, the two joint angles `t1` (the shoulder, measured from the horizontal) and `t2` (the elbow, relative to the first link). The tip is where the two links add up: `x = l1 cos t1 + l2`

`cos(t1 + t2)` and `y = l1 sin t1 + l2 sin(t1 + t2)`. That is **forward kinematics**, a function with one answer. The question a robot actually asks is the reverse: *given the point, what angles?* That is **inverse kinematics**, and it can have two answers (the elbow up or down) or none (the point is too far, or too close).

IN PYTHON	ON THE UNO IN C++
`math.acos(c)` raises `ValueError` if `c` is outside [−1, 1]	`acosf` returns `nan` and carries on; the function must *test* and say "unreachable" with a return value
a list of points, `for x, y in path:`	a fixed table of waypoints in flash or RAM, indexed by a leg counter
`time.sleep(0.02)` in a loop to move a servo a step	the scheduler of Programming for Robotics with Arduino C++ 1 calls the control task every 20 ms; nothing blocks
a servo library on a Linux board sets the angle in floats	`Servo::write(int)` takes whole degrees; `writeMicroseconds(int)` takes whole microseconds
`np.interp` or a spline for a table	a table of `uint16_t` in flash, read with `pgm_read_word` and interpolated by hand
any number of servos on any pins	one timer (Timer1) drives them; the pins you promised to something else are gone

2. Forward and inverse kinematics as functions

All of the chapter's code is in one header, `arm.h`: plain C++11 with no heap and no Arduino calls, so the same file runs on the PC and on the Uno. The inverse kinematics uses the law of cosines: `cos t2 = (x² + y² − l1² − l2²) / (2 l1 l2)`. If that is outside [−1, 1] there is no triangle, and the function returns `false` rather than a number; otherwise `t2` is plus or minus the angle whose cosine that is (the sign is the elbow up or down), and `t1` is the direction of the point minus the angle the first link makes inside the triangle. Here is the whole header; the parts after the inverse kinematics are used in the later sections:

```
// arm.h -- a two-link planar arm in plain functions (Programming for Robotics with Ardu
// Convention: shoulder angle t1 from the +x axis (counter-clockwise positive), elbow an
//   x = l1 cos(t1) + l2 cos(t1 + t2)       y = l1 sin(t1) + l2 sin(t1 + t2)
#pragma once
#include <stdint.h>
#include <math.h>
#ifdef __AVR__
```

```
#include <avr/pgmspace.h>
#else
#define PROGMEM
#define pgm_read_word(p) (*(p))
#endif

struct Arm { float l1, l2; };

inline void fk(const Arm& a, float t1, float t2, float& x, float& y) {
  x = a.l1 * cosf(t1) + a.l2 * cosf(t1 + t2);
  y = a.l1 * sinf(t1) + a.l2 * sinf(t1 + t2);
}

// ---- inverse kinematics with the library's acosf / atan2f ----
// law of cosines: cos(t2) = (x^2 + y^2 - l1^2 - l2^2) / (2 l1 l2). Outside [-1, 1] the p
// elbow_up: the elbow is above the line from the shoulder to the tip (t2 < 0 with this c
inline bool ik(const Arm& a, float x, float y, bool elbow_up, float& t1, float& t2) {
  float c2 = (x * x + y * y - a.l1 * a.l1 - a.l2 * a.l2) / (2.0f * a.l1 * a.l2);
  if (c2 > 1.0f || c2 < -1.0f) return false;
  float s2 = sqrtf(1.0f - c2 * c2);
  if (elbow_up) s2 = -s2;
  t2 = atan2f(s2, c2);                                          // = +-acos(c2)
  t1 = atan2f(y, x) - atan2f(a.l2 * s2, a.l1 + a.l2 * c2);
  return true;
}

// ---- the same, with lookup tables in flash instead of acosf / atan2f ----
// A table has n + 1 entries of 1e-4 radian (uint16_t) for a variable running over [0, 1
inline float table_interp(const uint16_t* t, uint8_t n, float u) {
  float pos = u * (float)n;
  int16_t i = (int16_t)pos;
  if (i < 0) i = 0;
  if (i > (int16_t)n - 1) i = (int16_t)n - 1;
  float frac = pos - (float)i;
  float v0 = (float)pgm_read_word(&t[i]), v1 = (float)pgm_read_word(&t[i + 1]);
  return (v0 + (v1 - v0) * frac) * 1.0e-4f;
}
inline float acos_lut(float c, const uint16_t* t, uint8_t n) {          // t[k] = acos(-1
  return table_interp(t, n, (c + 1.0f) * 0.5f);
}
inline float atan2_lut(float y, float x, const uint16_t* t, uint8_t n) { // t[k] = atan(k
  float ax = fabsf(x), ay = fabsf(y);
  if (ax == 0.0f && ay == 0.0f) return 0.0f;
  bool swap = ay > ax;
  float a = table_interp(t, n, swap ? ax / ay : ay / ax);
```

```
  if (swap) a = 1.5707964f - a;
  if (x < 0.0f) a = 3.1415927f - a;
  return y < 0.0f ? -a : a;
}
inline bool ik_lut(const Arm& a, float x, float y, bool elbow_up, float& t1, float& t2, 
  float c2 = (x * x + y * y - a.l1 * a.l1 - a.l2 * a.l2) / (2.0f * a.l1 * a.l2);
  if (c2 > 1.0f || c2 < -1.0f) return false;
  float s2 = sqrtf(1.0f - c2 * c2);
  if (elbow_up) s2 = -s2;
  t2 = acos_lut(c2, acos_t, n);
  if (elbow_up) t2 = -t2;
  t1 = atan2_lut(y, x, atan_t, n) - atan2_lut(a.l2 * s2, a.l1 + a.l2 * c2, atan_t, n);
  return true;
}

// ---- forward kinematics in integers, for servos that take whole degrees ----
// sine of a whole number of degrees: the table SIN_Q15 holds 91 entries for 0..90 degre
extern const uint16_t SIN_Q15[91] PROGMEM;
inline int16_t sin_q14(int16_t deg) {
  deg %= 360; if (deg < 0) deg += 360;
  bool neg = deg >= 180; if (neg) deg -= 180;
  if (deg > 90) deg = 180 - deg;
  int16_t v = (int16_t)(pgm_read_word(&SIN_Q15[deg]) >> 1);       // Q14 so that 1.0 fit
  return neg ? -v : v;
}
inline int16_t cos_q14(int16_t deg) { return sin_q14(deg + 90); }
inline void fk_int(int16_t l1_mm, int16_t l2_mm, int16_t t1_deg, int16_t t2_deg, int16_t
  int32_t x = (int32_t)l1_mm * cos_q14(t1_deg) + (int32_t)l2_mm * cos_q14(t1_deg + t2_de
  int32_t y = (int32_t)l1_mm * sin_q14(t1_deg) + (int32_t)l2_mm * sin_q14(t1_deg + t2_de
  x_tenth = (int16_t)((x * 10) >> 14); y_tenth = (int16_t)((y * 10) >> 14);
}

// ---- a trapezoidal speed profile: accelerate at a, cruise at v, brake at a; a short m
struct Trap {
  float d, v, a, ta, tc, T, sign;
  void plan(float dist, float vmax, float amax) {
    sign = dist < 0.0f ? -1.0f : 1.0f; d = fabsf(dist); a = amax;
    if (d == 0.0f) { v = 0.0f; ta = tc = T = 0.0f; return; }
    if (d < vmax * vmax / amax) { v = sqrtf(amax * d); ta = v / amax; tc = 0.0f; }
    else { v = vmax; ta = vmax / amax; tc = (d - vmax * ta) / vmax; }
    T = 2.0f * ta + tc;
  }
  float pos(float t) const {
    if (t <= 0.0f) return 0.0f;
    if (t >= T) return sign * d;
```

```
    float p;
    if (t < ta) p = 0.5f * a * t * t;
    else if (t < ta + tc) p = 0.5f * a * ta * ta + v * (t - ta);
    else { float r = T - t; p = d - 0.5f * a * r * r; }
    return sign * p;
  }
  float vel(float t) const {
    if (t <= 0.0f || t >= T) return 0.0f;
    float s;
    if (t < ta) s = a * t; else if (t < ta + tc) s = v; else s = a * (T - t);
    return sign * s;
  }
};
// two joints that start and finish together: the one that moves further sets the time,
inline void plan_sync(Trap& j0, Trap& j1, float d0, float d1, float vmax, float amax) {
  float m0 = fabsf(d0), m1 = fabsf(d1);
  if (m0 >= m1) { j0.plan(d0, vmax, amax); float s = m0 > 0.0f ? m1 / m0 : 0.0f; j1.plan
  else { j1.plan(d1, vmax, amax); float s = m1 / m0; j0.plan(d0, vmax * s, amax * s); }
}
```

The first experiment asks whether the two functions agree, and what the reachable region looks like. The arm has links of 100 and 80 mm (assumed). It can reach any point between a distance of 20 mm (the folded arm, `l1 - l2`) and 180 mm (the straight arm, `l1 + l2`) from the shoulder: a ring. Part B turns the angles into what a servo can say: whole degrees through `write(int)`, or whole microseconds through `writeMicroseconds` (10.31 µs per degree with the library's 544–2400 µs range, so 1 µs is 0.097°), and measures how far the tip lands from the asked-for point. Part C looks at the edge of the ring.

```
// e_kin.cpp -- forward and inverse kinematics of a 2-link arm, l1 = 100 mm, l2 = 80 mm
// Part A: ik() over a 1 mm grid of points in the 360 x 360 mm square around the shoulde
// Part B: a servo takes whole degrees (Servo::write(int)) or microseconds (writeMicrose
//         round the joint angles to those steps and measure how far the tip lands from
// Part C: the tip moves outward along y = 0 at 50 mm/s: the joint speeds that takes, ne
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "arm.h"

static const float D2R = 3.14159265f / 180.0f, R2D = 180.0f / 3.14159265f;

static float quant(float rad, float step_deg) { return std::round(rad * R2D / step_deg)
```

```
int main() {
  const Arm arm = {100.0f, 80.0f};
  std::printf("part A: ik() on a 1 mm grid, 361 x 361 points around the shoulder\n");
  for (int up = 1; up >= 0; --up) {
    long ok = 0; double worst = 0.0;
    for (int yi = -180; yi <= 180; ++yi)
      for (int xi = -180; xi <= 180; ++xi) {
        float t1, t2;
        if (!ik(arm, (float)xi, (float)yi, up != 0, t1, t2)) continue;
        ++ok;
        float x, y; fk(arm, t1, t2, x, y);
        double e = std::hypot(x - xi, y - yi); if (e > worst) worst = e;
      }
    std::printf("  elbow %-4s reachable %6ld points, worst error of fk(ik(p)) = %.5f mm\
  }
  std::printf("  area of the ring between r = 20 and r = 180 mm: %.0f mm^2 (the count of
  { float t1, t2; bool a = ik(arm, 181.0f, 0.0f, true, t1, t2), b = ik(arm, 19.0f, 0.0f,
    std::printf("  (181, 0): %s;  (19, 0): %s;  (180, 0): %s  (exactly straight)\n", a ?

  std::printf("\npart B: tip error from the servo's step (elbow up, working area y >= -4
  std::printf("  step                        mean error (mm)   worst error (mm)   points\
  struct { const char* name; float step; } modes[3] = {{"1 degree (write(int))", 1.0f}, {
  for (int m = 0; m < 3; ++m) {
    double sum = 0, worst = 0; long n = 0;
    for (int yi = -40; yi <= 170; ++yi)
      for (int xi = -170; xi <= 170; ++xi) {
        if (xi * xi + yi * yi > 170 * 170) continue;
        float t1, t2;
        if (!ik(arm, (float)xi, (float)yi, true, t1, t2)) continue;
        float x, y; fk(arm, quant(t1, modes[m].step), quant(t2, modes[m].step), x, y);
        double e = std::hypot(x - xi, y - yi); sum += e; if (e > worst) worst = e; ++n;
      }
    std::printf("  %-27s %15.3f   %16.3f   %6ld\n", modes[m].name, sum / n, worst, n);
  }

  std::printf("\npart C: tip moving outward along y = 0 at 50 mm/s: the joint speeds tha
  std::printf("  tip x (mm)   shoulder (deg/s)   elbow (deg/s)\n");
  for (float x : {40.0f, 100.0f, 150.0f, 170.0f, 178.0f, 179.5f, 179.95f}) {
    float a1 = 0, a2 = 0, b1 = 0, b2 = 0;
    ik(arm, x, 0.0f, true, a1, a2); ik(arm, x + 0.01f, 0.0f, true, b1, b2);
    std::printf("  %10.2f   %16.1f   %13.1f\n", x, std::fabs(b1 - a1) / 0.01f * 50.0f * 
  }
```

```
  return 0;
}
```

```
part A: ik() on a 1 mm grid, 361 x 361 points around the shoulder
  elbow up   reachable 100520 points, worst error of fk(ik(p)) = 0.00006 mm
  elbow down reachable 100520 points, worst error of fk(ik(p)) = 0.00006 mm
  area of the ring between r = 20 and r = 180 mm: 100531 mm^2 (the count of points shoul
  (181, 0): unreachable;  (19, 0): unreachable;  (180, 0): reachable  (exactly straight)

part B: tip error from the servo's step (elbow up, working area y >= -40 mm, r <= 170 mm
  step                        mean error (mm)   worst error (mm)   points
  1 degree (write(int))                 0.629              2.069    57780
  1 microsecond (0.097 deg)             0.061              0.199    57780
  0.5 us timer tick (0.048 deg)           0.030              0.100    57780

part C: tip moving outward along y = 0 at 50 mm/s: the joint speeds that needs
  tip x (mm)   shoulder (deg/s)   elbow (deg/s)
       40.00              23.6            37.7
      100.00              12.5            39.1
      150.00              24.4            58.1
      170.00              42.7            97.5
      178.00              95.6           215.6
      179.50             191.9           432.1
      179.95             637.0          1433.2
```

The functions agree: over 100,520 reachable points the round trip `fk(ik(p))` is within 0.00006 mm, with the elbow up or down, and the count of reachable points (100,520) is close to the area of the ring (100,531 mm^2). The points outside are refused: (181, 0) and (19, 0) are unreachable. The point (180, 0), exactly straight, is reported reachable, which depends on a floating-point value landing at 1.0 and not above it; an arm that is asked for its full length is asking for trouble anyway.

The servo's step is the arm's resolution. Whole degrees put the tip 0.63 mm off on average and 2.07 mm at worst, in the working area. Microseconds are ten times better (0.06 and 0.20 mm), and the library's own half-microsecond timer tick would be 0.03 and 0.10 mm. The tip error is roughly the reach times the angular step, so it is worst where the arm is longest, and a Python script that sets "angle = 47.3" on a PC has hidden that the servo would have taken 47.

The edge of the workspace is a trap. To move the tip outward along a line at 50 mm/s the joints need 12–58°/s between 40 and 150 mm from the shoulder, 43 and 98°/s at 170 mm, and

637 and 1,433°/s at 179.95 mm, a few hundredths of a millimetre before the arm is straight. An arm that is told to reach a point near its full length must either slow down to a crawl or fail; a planner keeps its waypoints away from the singularity.

3. What the Servo library does

I installed the Servo library (version 1.3.0) and read its source, because the chapter's claims about it are claims about code. What it does, with the lines it comes from:

- **The pulse range.** `MIN_PULSE_WIDTH 544` and `MAX_PULSE_WIDTH 2400` in `Servo.h`; `write(int value)` for 0–180 does `map(value, 0, 180, SERVO_MIN(), SERVO_MAX())` in `avr/Servo.cpp`. So an angle in whole degrees is 10.31 µs per degree, rounded by integer `map`.
- **The clock.** Timer1 with a prescaler of 8 (`TCCR1B = _BV(CS11)`): a tick is 0.5 µs at 16 MHz, so a pulse width is held to half a microsecond. `writeMicroseconds(int)` takes whole microseconds, and clips the value to the (trimmable) range.
- **The refresh.** `REFRESH_INTERVAL 20000`: a 20 ms frame. The servos on one timer are pulsed *one after the other* inside it, up to 12 per timer (`SERVOS_PER_TIMER 12`), by an interrupt on Timer1 that calls `digitalWrite`. A command written during a frame takes effect at the servo's next pulse, so there is no point commanding faster than 50 Hz.
- **The timer it takes.** `attach` sets `TCCR1A = 0` (normal counting mode) on the first servo. On an Uno, pins 9 and 10 are the outputs of Timer1's hardware PWM (the pin table of Programming for Robotics with Arduino C++ 2), so from the first `attach` I expect `analogWrite` on 9 and 10 to stop working. That follows from the source and the pin table; *I did not run it*. The robot's motors are on pins 5 and 6 (Timer0), which are not touched.

Two consequences are measurable without a board. The pin budget: the plan of Programming for Robotics with Arduino C++ 2 uses every pin except 9 and 10 (0 and 1 are the USB link, 18 and 19 the I2C bus of Programming for Robotics with Arduino C++ 10), so an arm with *three* servos (shoulder, elbow, gripper) must give up a pin: `robot_ch11.ino` takes pin 13, the LED, for the gripper, because the Servo library will drive any digital pin. And the cost in bytes, in section 5.

4. A trapezoidal speed profile

A servo commanded straight to a new angle moves as fast as it can, and a real arm shakes when it is slammed. The usual remedy is a **trapezoidal profile**: accelerate at a constant rate, cruise at the top speed, brake at the same rate. The position at time `t` is a pair of parabolas and a line; `Trap` in `arm.h` computes it with no loop. A short move never reaches the top speed and becomes a triangle: the peak speed is `sqrt(a d)`. Two joints that must arrive together are *synchronised*: the joint with the longer move sets the time, and the other runs the same shape with its speed and acceleration limits scaled by the ratio of the distances. The experiment (the limits of 120°/s and 400°/s² are assumed) checks the maths numerically, shows what a servo that takes whole degrees does to a slow profile, and synchronises two joints:

```
// e_trap.cpp -- trapezoidal speed profiles for a joint, and what a servo that takes whol
// Part A: moves of several lengths: acceleration time, cruise time, total time, peak sp
// Part B: a slow move of 30 degrees at 20 deg/s, commanded at 50 Hz (every 20 ms), roun
// Part C: two joints moving 60 and 15 degrees, synchronised: both finish together.
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "arm.h"

int main() {
  const float VMAX = 120.0f, AMAX = 400.0f;
  std::printf("part A: one joint, vmax %.0f deg/s, amax %.0f deg/s^2\n", VMAX, AMAX);
  std::printf("  move (deg)   accelerate (s)   cruise (s)   total (s)   peak speed (deg/
  for (float d : {5.0f, 10.0f, 20.0f, 36.0f, 40.0f, 90.0f, 180.0f}) {
    Trap t; t.plan(d, VMAX, AMAX);
    double integ = 0, amax_seen = 0; float prev_v = 0.0f;
    for (int i = 1; i <= (int)(t.T * 1000.0f) + 1; ++i) {
      float tt = i * 0.001f, v = t.vel(tt);
      integ += v * 0.001; double a = std::fabs(v - prev_v) / 0.001; if (a > amax_seen) a
    }
    std::printf("  %10.0f   %14.3f   %10.3f   %9.3f   %18.1f   %-9s   %23.3f   %30.0f\n"
  }

  std::printf("\npart B: a move of 30 degrees at 20 deg/s (acceleration 400), a new comm
  std::printf("  command rounded to   updates   updates with no motion   updates with 1
  for (int mode = 0; mode < 2; ++mode) {
    Trap t; t.plan(30.0f, 20.0f, AMAX);
    int n = 0, zero = 0, one = 0; double fastest = 0, slowest = 1e9; double prev = 0.0;
    for (int i = 1; (i * 0.02f) <= t.T + 0.0001f; ++i) {
      double p = t.pos(i * 0.02f);
      double q = mode == 0 ? std::round(p) : std::round(p * 10.0) / 10.0;
      double step = q - prev; prev = q; ++n;
```

```
      double sp = std::fabs(step) / 0.02;
      if (step == 0.0) ++zero; else if (std::fabs(step - 1.0) < 1e-9) ++one;
      if (sp > fastest) fastest = sp;
      if (step != 0.0 && sp < slowest) slowest = sp;
    }
    std::printf("  %-18s %8d   %22d   %18d   %43.1f   %23.1f\n", mode == 0 ? "whole degr
  }

  std::printf("\npart C: two joints, 60 and 15 degrees, synchronised (vmax %.0f, amax %.
  { Trap a, b; plan_sync(a, b, 60.0f, 15.0f, VMAX, AMAX);
    std::printf("  joint 0: T = %.4f s, peak %.1f deg/s;  joint 1: T = %.4f s, peak %.1f
    for (float t : {0.0f, 0.1f, 0.2f, 0.3f, 0.4f, a.T}) std::printf("  t = %.3f s   join
  return 0;
}
```

```
part A: one joint, vmax 120 deg/s, amax 400 deg/s^2
  move (deg)   accelerate (s)   cruise (s)   total (s)   peak speed (deg/s)   shape
           5            0.112        0.000       0.224                 44.7   triangle
          10            0.158        0.000       0.316                 63.2   triangle
          20            0.224        0.000       0.447                 89.4   triangle
          36            0.300        0.000       0.600                120.0   triangle
          40            0.300        0.033       0.633                120.0   trapezoid
          90            0.300        0.450       1.050                120.0   trapezoid
         180            0.300        1.200       1.800                120.0   trapezoid

part B: a move of 30 degrees at 20 deg/s (acceleration 400), a new command every 20 ms
  command rounded to   updates   updates with no motion   updates with 1 deg   fastest s
  whole degrees            77                       47                   30
  0.1 degree               77                        0                    0

part C: two joints, 60 and 15 degrees, synchronised (vmax 120, amax 400 for the longer o
  joint 0: T = 0.8000 s, peak 120.0 deg/s;  joint 1: T = 0.8000 s, peak 30.0 deg/s
  t = 0.000 s   joint 0 at   0.000 deg   joint 1 at   0.000 deg   (fraction 0.0000 and 0
  t = 0.100 s   joint 0 at   2.000 deg   joint 1 at   0.500 deg   (fraction 0.0333 and 0
  t = 0.200 s   joint 0 at   8.000 deg   joint 1 at   2.000 deg   (fraction 0.1333 and 0
  t = 0.300 s   joint 0 at  18.000 deg   joint 1 at   4.500 deg   (fraction 0.3000 and 0
  t = 0.400 s   joint 0 at  30.000 deg   joint 1 at   7.500 deg   (fraction 0.5000 and 0
  t = 0.800 s   joint 0 at  60.000 deg   joint 1 at  15.000 deg   (fraction 1.0000 and 1
```

The profile is exact: the integral of the speed is the distance in every case, and the largest acceleration found by differences is the limit. The break between a triangle and a trapezoid is at 36° (`vmax² / amax`). Part B is the unexpected one. A move of 30° at 20°/s, commanded every 20

ms and rounded to whole degrees, becomes 47 updates in which nothing moves and 30 in which the joint jumps 1°: a speed of 50°/s, two and a half times the one asked for, in a stop-and-go pattern. Rounded to 0.1° (a bit more than 1 µs) the speed varies smoothly between 5 and 20°/s. A slow move is where whole degrees hurt, and `writeMicroseconds` with a float angle is the remedy. Part C: the joints that move 60° and 15° arrive together at 0.800 s, and at every time the fractions of the distance covered are equal (0.0333 and 0.0333 at 0.1 s): the joints start and stop together.

5. Lookup tables in flash

The inverse kinematics needs three angles: the elbow (an arc-cosine, or the arc-tangent of its sine and cosine, as `ik()` does) and two directions (arc-tangents). On a chip with no floating-point unit they are library routines, and an obvious idea is to replace them by a table: precompute the function at evenly spaced points, store it in flash, and interpolate. `ik_lut()` in `arm.h` does this: `acos` over [−1, 1] and `atan` over [0, 1] (the octant is folded back by hand), 64 steps each, values in units of 10^{-4} rad in `uint16_t`. The tables are made on the PC and written as C++ source, so there is nothing to compute at start-up, and on the AVR they are `PROGMEM` and take no RAM (the hard lesson of Programming for Robotics with Arduino C++ 9: a `const` array is RAM unless it says otherwise).

```
// e_gen.cpp -- writes tables.h: the lookup tables of arm.h as C++ source, generated on
// ACOS_T: acos(-1 + 2 k / 64) in units of 1e-4 rad, 65 entries; ATAN_T: atan(k / 64) in
#include <cstdio>
#include <cmath>
#include "arm.h"

int main() {
  FILE* f = std::fopen("tables.h", "w");
  std::fprintf(f, "// tables.h -- generated by e_gen.cpp: do not edit. Lookup tables for
  std::fprintf(f, "const uint8_t LUT_N = 64;\n");
  std::fprintf(f, "const uint16_t ACOS_T[65] PROGMEM = {");
  for (int k = 0; k <= 64; ++k) std::fprintf(f, "%s%ld", k ? ", " : "", std::lround(std:
  std::fprintf(f, "};\nconst uint16_t ATAN_T[65] PROGMEM = {");
  for (int k = 0; k <= 64; ++k) std::fprintf(f, "%s%ld", k ? ", " : "", std::lround(std:
  std::fprintf(f, "};\nconst uint16_t SIN_Q15[91] PROGMEM = {");
  for (int d = 0; d <= 90; ++d) std::fprintf(f, "%s%ld", d ? ", " : "", std::lround(std:
  std::fprintf(f, "};\n");
  std::fclose(f);
  std::printf("tables.h written: ACOS_T and ATAN_T of 65 entries (130 bytes each), SIN_Q
```

```
  return 0;
}
```

```
tables.h written: ACOS_T and ATAN_T of 65 entries (130 bytes each), SIN_Q15 of 91 entrie
```

How big must a table be? The experiment builds tables of several sizes and runs `ik_lut()` over the whole ring, feeding the result to the ordinary `fk()`:

```
// e_lut.cpp -- inverse kinematics with lookup tables instead of acosf and atan2f: how b
// For each table size n the tables are built here at run time and ik_lut() from arm.h i
// to the ordinary fk(), and the distance from the asked-for point is the tip error. The
// Then, for n = 64, where the error sits: by the distance of the point from the shoulde
#include <cstdio>
#include <cmath>
#include <vector>
#include <initializer_list>
#include "arm.h"

static void build(int n, std::vector<uint16_t>& ac, std::vector<uint16_t>& at) {
  ac.resize(n + 1); at.resize(n + 1);
  for (int k = 0; k <= n; ++k) { ac[k] = (uint16_t)std::lround(std::acos(-1.0 + 2.0 * k
}

int main() {
  const Arm arm = {100.0f, 80.0f};
  std::printf("tip error of ik_lut() against the table size, over the reachable 1 mm gri
  std::printf("  entries n   flash (bytes)   mean error (mm)   worst error (mm)   worst
  std::vector<uint16_t> ac, at;
  for (int n : {8, 16, 32, 64, 128}) {
    build(n, ac, at);
    double sum = 0, worst = 0, worst170 = 0, worst_r = 0; long cnt = 0;
    for (int yi = -180; yi <= 180; ++yi)
      for (int xi = -180; xi <= 180; ++xi) {
        float t1 = 0, t2 = 0;
        if (!ik_lut(arm, (float)xi, (float)yi, true, t1, t2, ac.data(), at.data(), (uint
        float x, y; fk(arm, t1, t2, x, y);
        double e = std::hypot(x - xi, y - yi), r = std::hypot((double)xi, (double)yi);
        sum += e; ++cnt;
        if (e > worst) { worst = e; worst_r = r; }
        if (r >= 40.0 && r <= 170.0 && e > worst170) worst170 = e;
      }
    std::printf("  %9d   %13d   %15.4f   %16.3f   %32.3f   %15.1f\n", n, 2 * 2 * (n + 1)
```

```
  }

  std::printf("\nn = 64: the worst tip error in rings of the distance r from the shoulde
  std::printf("  r from-to (mm)   points   mean error (mm)   worst error (mm)\n");
  build(64, ac, at);
  const double edges[] = {20, 40, 80, 120, 160, 170, 175, 178, 180.5};
  for (int b = 0; b < 8; ++b) {
    double sum = 0, worst = 0; long cnt = 0;
    for (int yi = -180; yi <= 180; ++yi)
      for (int xi = -180; xi <= 180; ++xi) {
        double r = std::hypot((double)xi, (double)yi);
        if (r < edges[b] || r >= edges[b + 1]) continue;
        float t1 = 0, t2 = 0;
        if (!ik_lut(arm, (float)xi, (float)yi, true, t1, t2, ac.data(), at.data(), 64)) {
        float x, y; fk(arm, t1, t2, x, y);
        double e = std::hypot(x - xi, y - yi); sum += e; ++cnt; if (e > worst) worst = e;
      }
    std::printf("  %5.0f - %-6.1f   %6ld   %15.4f   %16.3f\n", edges[b], edges[b + 1], c
  }
  return 0;
}
```

```
tip error of ik_lut() against the table size, over the reachable 1 mm grid (elbow up), r
  entries n   flash (bytes)   mean error (mm)   worst error (mm)   worst error for 40 <=
         8              36           2.5255             14.106
        16              68           0.9348              9.982
        32             132           0.3404              7.063
        64             260           0.1236              4.992
       128             516           0.0455              3.534

n = 64: the worst tip error in rings of the distance r from the shoulder
  r from-to (mm)   points   mean error (mm)   worst error (mm)
     20 - 40.0       3768           1.4981              4.992
     40 - 80.0      15056           0.0300              0.158
     80 - 120.0     25144           0.0052              0.014
    120 - 160.0     35156           0.0049              0.021
    160 - 170.0     10380           0.0117              0.041
    170 - 175.0      5440           0.0331              0.101
    175 - 178.0      3276           0.1205              0.358
    178 - 180.5      2300           2.3163              4.988
```

The error falls fast with the table size (mean 2.5 mm at 8 entries, 0.12 mm at 64, 0.05 at 128) but it is **not spread evenly**. For n = 64 the points between 40 and 170 mm from the shoulder are accurate to 0.16 mm at worst, but the two ends are bad: 5 mm near the folded arm (r below 40) and 5 mm near the straight arm (r above 178). The reason is the shape of the function: `acos` has an infinite slope at ±1, exactly where the arm is folded or straight, so a table with even steps cannot follow it there, and these are the two places where a small angle error moves the tip most. A table is only as good as the region you promise to stay in; the table size is the price of how close to the edge you are willing to go.

What the tables cost on the Uno

The cost-test sketch has eight variants, one function each, fed from `analogRead` so the compiler cannot remove it, and printed as an integer so Serial's float printing is in none of them (the lesson of Programming for Robotics with Arduino C++ 10). Variant 0 is the baseline.

```
// s_arm.ino -- cost test for chapter 11: the arm functions on the Uno. Compile-checked (
// VARIANT 0: baseline, 1: ik() with acosf/atan2f, 2: ik_lut() with the tables in flash,
//            5: Trap::pos (one step of a trapezoid), 6: Trap::plan, 7: two Servo objects
#include "tables.h"
#if VARIANT == 7
#include <Servo.h>
#endif

#ifndef VARIANT
#define VARIANT 1
#endif

static const Arm arm = {100.0f, 80.0f};

#if VARIANT == 1
__attribute__((noinline)) int32_t step(float x, float y) { float t1, t2; if (!ik(arm, x,
#elif VARIANT == 2
__attribute__((noinline)) int32_t step(float x, float y) { float t1, t2; if (!ik_lut(arm
#elif VARIANT == 3
__attribute__((noinline)) int32_t step(float t1, float t2) { float x, y; fk(arm, t1, t2,
#elif VARIANT == 4
__attribute__((noinline)) int32_t step(int16_t t1, int16_t t2) { int16_t x, y; fk_int(10
#elif VARIANT == 5
static Trap trap;
__attribute__((noinline)) int32_t step(float t) { return (int32_t)(trap.pos(t) * 100.0f)
#elif VARIANT == 6
__attribute__((noinline)) int32_t step(float d, float v) { Trap t; t.plan(d, v, 400.0f);
```

```
#elif VARIANT == 7
static Servo s1, s2;
#endif

void setup() {
  Serial.begin(115200);
#if VARIANT == 5
  trap.plan((float)analogRead(A2) * 0.1f, 120.0f, 400.0f);
#elif VARIANT == 7
  s1.attach(9); s2.attach(10);
#endif
}

void loop() {
  int a = analogRead(A0), b = analogRead(A1);      // inputs the compiler cannot know
#if VARIANT == 0
  Serial.println((int32_t)(a + b) * 1000L);
#elif VARIANT == 1 || VARIANT == 2
  Serial.println(step((float)a * 0.18f, (float)b * 0.18f - 40.0f));
#elif VARIANT == 3
  Serial.println(step((float)a * 0.003f, (float)b * 0.003f));
#elif VARIANT == 4
  Serial.println(step((int16_t)(a / 8), (int16_t)(b / 8)));
#elif VARIANT == 5
  Serial.println(step((float)a * 0.001f)); (void)b;
#elif VARIANT == 6
  Serial.println(step((float)a * 0.1f, (float)b * 0.1f + 10.0f));
#elif VARIANT == 7
  s1.write(a / 6); s2.write(b / 6);
  Serial.println((int32_t)(a + b) * 1000L);
#endif
  delay(20);
}
```

```
variant                                flash (bytes)   vs baseline   RAM (bytes)   vs
0  baseline (nothing)                       1960             -            188
1  ik: sqrtf and atan2f (the library)       4082         +2122            188
2  ik_lut: tables in flash                  4552         +2592            188
3  fk: sinf and cosf                        3412         +1452            188
4  fk_int: integer sine table               2548          +588            188
5  Trap::pos (one step)                     4220         +2260            216
```

```
6  Trap::plan                              3548        +1588          188
7  two Servo objects (attach, write)       3592        +1632          232
```

And, from `avr-objdump` (script `count_arm.py`), the step function of each variant: its size, its instructions, and the library routines it calls. These are the function itself; in variant 2 two helpers, `table_interp` and `atan2_lut`, were not inlined and are not counted in its 188 instructions:

```
ik: acosf, atan2f, sqrtf          402 bytes  180 instructions   calls: __addsf3 x3, __cm
ik_lut: tables + interpolation    422 bytes  188 instructions   calls: _Z12table_interpP
fk: sinf, cosf                    294 bytes  133 instructions   calls: __addsf3 x4, __fi
fk_int: integer sine table        204 bytes   92 instructions   calls: _Z7sin_q14i x4, _
Trap::pos (one step)              492 bytes  194 instructions   calls: __addsf3 x2, __cm
Trap::plan                        254 bytes  112 instructions   calls: __addsf3 x2, __cm
```

And the sizes of the routines reached, from `avr-nm` (hexadecimal bytes):

```
$ avr-nm -S -C: symbol sizes in hex bytes (static code size, not run time)
-- variant 1 (ik) --
00000f5e 0000001c T __fp_mpack
00000cf8 0000004e T __fp_powser
00000d84 00000044 T __fp_split3
00000a0c 00000086 T atan2
00000ed6 0000008c T sqrt
-- variant 3 (fk) --
00000a80 00000056 T __fp_rempio2
00000af4 00000028 T __fp_sinus
0000095e 0000000a T cos
00000c48 00000010 T sin
-- variant 2 (ik_lut): the two helpers that are not inlined, and the tables --
000001c2 000000e6 t table_interp(unsigned int const*, unsigned char, float) [clone .cons
000002a8 00000150 t atan2_lut(float, float, unsigned int const*, unsigned char) [clone .
00000068 00000082 t ACOS_T
000000ea 00000082 t ATAN_T
```

The result is the opposite of what I expected. **The lookup tables did not save flash: they cost 470 bytes more** (+2592 against +2122). The library's `atan2` is only 134 bytes (0x86) and `sqrt` 140, so the routine the table replaces was small; the replacement is two helpers of 230 and 336 bytes plus 260 bytes of tables, and all the float multiplications and the division stay (the inverse kinematics needs them for the law of cosines whichever way the angles are found). Whether the tables are *faster* I cannot say: nothing was timed, and the idea that a table is faster than a long library routine is plausible and not shown here. What the numbers do show is that a table is not

a free saving of space, and that the tables only make sense if the time matters and the flash is there.

The *integer* table is different. `fk_int()` takes whole degrees, which is all the servo can use anyway, looks the sine up in a 91-entry table (182 bytes, 0 to 90°; the other quadrants are folded back), and does everything in integers: 92 instructions and no float call at all, +588 bytes of flash against +1452 for `fk()` with `sinf` and `cosf`. When the answer is only needed to the servo's resolution, the table with integer arithmetic is the cheap one: not because lookup is clever, but because it removes the floating-point library.

6. A pick-and-place cycle on the simulator

Now the pieces together. The shoulder is at the origin, the table surface at y = −40 mm, the object is at a pick point, the grip point is lowered to 2 mm above the table, and the arm carries the object to a place point. The cycle has seven legs: from *home* to above the pick, down to the pick, up, across to above the place, down, up, home. Each leg is a joint-space move of the previous section, commanded every 20 ms. The servo model is an assumption: it takes whole degrees (or 1 µs steps), follows the command with a straight ramp between updates, and has 0.8° of play: it does not move until the command has gone 0.4° past it. The tip is followed every millisecond.

```
// e_pick.cpp -- a pick-and-place cycle for the 2-link arm on the simulator (host build;
// Arm l1 = 100, l2 = 80 mm, shoulder at the origin, table surface at y = -40 mm. The gr
// Sequence: home (110, 40) -> above pick -> pick -> above pick -> above place -> place
// commanded every 20 ms. The servos take whole degrees (write(int)) or 1 us steps (0.09
// The tip is followed every millisecond (the command between two updates is a straight
#include <cstdio>
#include <cmath>
#include <vector>
#include "arm.h"
#include "sim.h"

static const float D2R = 3.14159265f / 180.0f, R2D = 180.0f / 3.14159265f;
struct Pt { float x, y; };
struct Leg { Pt to; bool is_down; };

struct Result { double total_s, deepest, err_pick, err_place, sideways; int hits; };

static Result run_cycle(const Arm& arm, float px, float qx, float step_deg, float grasp_
```

```
  const float TRAVEL = 20.0f, GRASP = grasp_y, TABLE = -40.0f, PLAY = 0.8f;
  Leg legs[7] = {{{px, TRAVEL}, false}, {{px, GRASP}, true}, {{px, TRAVEL}, false}, {{qx
  Result r = {0, 0, 0, 0, 0, 0};
  float j1 = 0, j2 = 0; ik(arm, 110.0f, 40.0f, true, j1, j2);                    // jo
  float act1 = j1, act2 = j2;                                                  // where the
  float prev_c1 = j1, prev_c2 = j2;                                            // the last
  double clock = 0;
  for (int li = 0; li < 7; ++li) {
    float t1, t2;
    if (!ik(arm, legs[li].to.x, legs[li].to.y, true, t1, t2)) { r.total_s = -1; return r
    Trap a, b; plan_sync(a, b, (t1 - j1) * R2D, (t2 - j2) * R2D, 120.0f, 400.0f);
    double leg_t = a.T > b.T ? a.T : b.T;
    int updates = (int)std::ceil(leg_t / 0.02 - 1e-9);
    double x_at_start = 0, x_sum = 0; float x_min = 1e9f, x_max = -1e9f;
    float q1_prev = prev_c1, q2_prev = prev_c2;
    for (int u = 1; u <= updates; ++u) {
      double tt = u * 0.02; if (tt > leg_t) tt = leg_t;
      float c1 = j1 + a.pos((float)tt) * D2R, c2 = j2 + b.pos((float)tt) * D2R;
      float q1 = std::round(c1 * R2D / step_deg) * step_deg * D2R, q2 = std::round(c2 *
      double dt = tt - (u - 1) * 0.02;                                          // length o
      int ms = (int)std::lround(dt * 1000.0);
      for (int m = 1; m <= ms; ++m) {
        float f = (float)m / (float)ms;
        float cmd1 = q1_prev + (q1 - q1_prev) * f, cmd2 = q2_prev + (q2 - q2_prev) * f;
        float h = 0.5f * PLAY * D2R;
        if (cmd1 > act1 + h) act1 = cmd1 - h; else if (cmd1 < act1 - h) act1 = cmd1 + h;
        if (cmd2 > act2 + h) act2 = cmd2 - h; else if (cmd2 < act2 - h) act2 = cmd2 + h;
        float x, y; fk(arm, act1, act2, x, y);
        if (y < TABLE) { r.hits++; if (TABLE - y > r.deepest) r.deepest = TABLE - y; }
        if (legs[li].is_down) { if (x < x_min) x_min = x; if (x > x_max) x_max = x; }
      }
      q1_prev = q1; q2_prev = q2;
      (void)x_at_start; (void)x_sum;
    }
    prev_c1 = q1_prev; prev_c2 = q2_prev;
    float x, y; fk(arm, act1, act2, x, y);
    double err = std::hypot(x - legs[li].to.x, y - legs[li].to.y);
    if (li == 1) r.err_pick = err;
    if (li == 4) r.err_place = err;
    if (legs[li].is_down && x_max - x_min > r.sideways) r.sideways = x_max - x_min;
    if (verbose) std::printf("  leg %d to (%6.1f, %6.1f)   %.3f s   %3d updates   tip en
    clock += updates * 0.02; r.total_s = clock;
    j1 = t1; j2 = t2;
  }
  return r;
```

```
}

int main() {
  const Arm arm = {100.0f, 80.0f};
  std::printf("one cycle: pick at x = 120, place at x = 60 (servo step 1 degree)\n");
  run_cycle(arm, 120.0f, 60.0f, 1.0f, -38.0f, true);
  struct { const char* name; float step; } modes[2] = {{"whole degrees", 1.0f}, {"1 micr
  const float grasps[2] = {-38.0f, -39.5f};
  for (int g = 0; g < 2; ++g) {
    std::printf("\n200 cycles, pick and place x drawn from 60..140 mm (at least 30 mm ap
    std::printf("  servo step        cycle time (s) mean / max   tip below the table: cy
    for (int m = 0; m < 2; ++m) {
      Rng rng(4242u);
      double tsum = 0, tmax = 0, deep = 0, pe = 0, pw = 0, qe = 0, qw = 0, sd = 0, sw = 
      while (done < 200) {
        float px = 60.0f + 80.0f * (float)rng.uniform(), qx = 60.0f + 80.0f * (float)rng
        if (std::fabs(px - qx) < 30.0f) continue;
        Result r = run_cycle(arm, px, qx, modes[m].step, grasps[g], false);
        if (r.total_s < 0) continue;
        ++done; ++cycles; tsum += r.total_s; if (r.total_s > tmax) tmax = r.total_s;
        if (r.hits) ++hit_cycles;
        if (r.deepest > deep) deep = r.deepest;
        pe += r.err_pick; if (r.err_pick > pw) pw = r.err_pick;
        qe += r.err_place; if (r.err_place > qw) qw = r.err_place;
        sd += r.sideways; if (r.sideways > sw) sw = r.sideways;
      }
      std::printf("  %-16s  %12.2f / %-6.2f   %18d / %-18.2f   %18.2f / %-8.2f   %19.2f 
    }
  }
  return 0;
}
```

```
one cycle: pick at x = 120, place at x = 60 (servo step 1 degree)
  leg 1 to ( 120.0,   20.0)   0.347 s    18 updates   tip ends at ( 119.32,   19.50), 0.
  leg 2 to ( 120.0,  -38.0)   0.534 s    27 updates   tip ends at ( 119.82,  -37.07), 0.
  leg 3 to ( 120.0,   20.0)   0.534 s    27 updates   tip ends at ( 120.38,   18.61), 1.
  leg 4 to (  60.0,   20.0)   0.676 s    34 updates   tip ends at (  60.23,   20.20), 0.
  leg 5 to (  60.0,  -38.0)   0.728 s    37 updates   tip ends at (  59.92,  -37.65), 0.
  leg 6 to (  60.0,   20.0)   0.728 s    37 updates   tip ends at (  60.23,   20.20), 0.
  leg 7 to ( 110.0,   40.0)   0.642 s    33 updates   tip ends at ( 109.37,   39.37), 0.

200 cycles, pick and place x drawn from 60..140 mm (at least 30 mm apart); table at y =
  servo step        cycle time (s) mean / max   tip below the table: cycles / deepest (m
  whole degrees             3.93 / 4.44                      0 / 0.00
```

```
    1 microsecond             3.93 / 4.44                    0 / 0.00

  200 cycles, pick and place x drawn from 60..140 mm (at least 30 mm apart); table at y =
    servo step        cycle time (s) mean / max   tip below the table: cycles / deepest (m
    whole degrees             3.96 / 4.44                    3 / 0.34
    1 microsecond             3.96 / 4.44                    0 / 0.00
```

One cycle (pick at x = 120, place at x = 60) is seven legs of 0.35–0.73 s. Over 200 cycles with random pick and place points the commanded motion takes 3.9 s (4.4 s at worst), the same with either servo step; the gripper's pauses are not counted. Three results matter.

Play, not the step, is the floor. With 1 µs steps the tip is still 0.70 mm from the target on average (0.8 mm at worst): the 0.8° of play of the model, which no finer command removes. Whole degrees add a worse error (0.94 mm mean, up to 2.1 mm) on top.

The margin above the table. With 2 mm of clearance nobody touches the table, though the error of whole degrees reaches 2.07 mm (the error happened to point sideways). At 0.5 mm of clearance, 3 cycles in 200 with whole degrees push the tip as much as 0.34 mm *below* the table, and none with microseconds. A clearance smaller than the largest error is not a clearance; on a real arm I would choose it from a measured error, not a simulated one.

Joint-space legs are not straight. Lowering the tip "straight down" at the pick moves it sideways by 5.2–5.7 mm on average and up to 6.5–8.0 mm in the worst cycle, because the two joints move together by their own proportions and the tip follows the curve that makes. For picking something up from a hole or between two others, that is a collision. Challenge 3 compares this with a straight-line move.

7. On the Uno

`robot_ch11.ino` runs the same cycle: seven legs of the table `LEGS`; for each leg `ik_lut()` gives the joint angles (refusing a point that is out of reach or that a servo cannot reach, with the zero positions below), `plan_sync()` sets up the two profiles, and a 20 ms control task writes `writeMicroseconds` with the interpolated angles; the gripper closes after leg 2 and opens after leg 5, with a 400 ms pause each time. The servo's zero positions are assumptions that I chose so that every point of the cycle lies inside 0–180° of each servo (shoulder −60 to +120°, elbow −150 to +30°): the cycle uses 17–86° at the shoulder and −141 to −73° at the elbow, so an

elbow servo centred at 90° (±90°) could not do the job. The scheduler is the one of Programming for Robotics with Arduino C++ 1.

```
// robot_ch11.ino -- Programming for Robotics with Arduino C++, chapter 11: a pick-and-p
// Compile-checked for the Uno (arduino:avr:uno). NOT run on a board: no arm was built o
// measure your own arm and change them. The inverse kinematics uses the lookup tables i
#include <Servo.h>
#include "scheduler.h"
#include "pins.h"
#include "tables.h"

const uint8_t PIN_SHOULDER = 9, PIN_ELBOW = 10, PIN_GRIP = 13;     // the only free pins
static_assert(pin_ok(PIN_SHOULDER, R_DIGITAL) && pin_ok(PIN_ELBOW, R_DIGITAL) && pin_ok(
// The Servo library takes Timer1: analogWrite on pins 9 and 10 stops working (the motor

const Arm ARM = {100.0f, 80.0f};                                   // ASSUMED link lengt
const float SHOULDER_ZERO = 60.0f, ELBOW_ZERO = 150.0f;            // ASSUMED: servo ang
const float V_MAX = 120.0f, A_MAX = 400.0f;                        // ASSUMED deg/s and
const float TRAVEL_Y = 20.0f, GRASP_Y = -38.0f, PICK_X = 120.0f, PLACE_X = 60.0f;
const int GRIP_OPEN = 40, GRIP_CLOSED = 110;                       // ASSUMED servo degr

struct Wp { float x, y; };
static const Wp LEGS[7] = {{PICK_X, TRAVEL_Y}, {PICK_X, GRASP_Y}, {PICK_X, TRAVEL_Y}, {P

static Servo s_shoulder, s_elbow, s_grip;
static float cur1 = 0.0f, cur2 = 0.0f;                             // joint angles at th
static float to1 = 0.0f, to2 = 0.0f;
static Trap tj1, tj2;
static uint8_t leg = 0;
static uint32_t leg_start = 0, dwell_until = 0;
static bool running = false, failed = false;
static Scheduler<2> sch;

static int deg_to_us(float servo_deg) {                            // the Servo library'
  if (servo_deg < 0.0f) servo_deg = 0.0f; else if (servo_deg > 180.0f) servo_deg = 180.0
  return 544 + (int)(servo_deg * (1856.0f / 180.0f) + 0.5f);
}
static void write_joints(float d1, float d2) { s_shoulder.writeMicroseconds(deg_to_us(SH

static bool start_leg(uint8_t i, uint32_t now) {
  float t1, t2;
  if (!ik_lut(ARM, LEGS[i].x, LEGS[i].y, true, t1, t2, ACOS_T, ATAN_T, LUT_N)) return fa
  to1 = t1 * 57.29578f; to2 = t2 * 57.29578f;
  float s1 = SHOULDER_ZERO + to1, s2 = ELBOW_ZERO + to2;
```

```
  if (s1 < 0.0f || s1 > 180.0f || s2 < 0.0f || s2 > 180.0f) return false;
  plan_sync(tj1, tj2, to1 - cur1, to2 - cur2, V_MAX, A_MAX);
  leg_start = now;
  return true;
}

static void control_task(void*, uint32_t now) {
  if (!running || failed) return;
  if ((int32_t)(now - dwell_until) < 0) return;                    // waiting for the gr
  float t = (float)(now - leg_start) * 0.001f;
  float T = tj1.T > tj2.T ? tj1.T : tj2.T;
  if (t >= T) {                                                    // the leg is done: l
    cur1 = to1; cur2 = to2; write_joints(cur1, cur2);
    if (leg == 1) { s_grip.write(GRIP_CLOSED); dwell_until = now + 400; }
    if (leg == 4) { s_grip.write(GRIP_OPEN); dwell_until = now + 400; }
    if (++leg >= 7) { running = false; return; }
    if (!start_leg(leg, now + (dwell_until > now ? 400 : 0))) { failed = true; }
    return;
  }
  write_joints(cur1 + tj1.pos(t), cur2 + tj2.pos(t));
}
static void report_task(void*, uint32_t now) {
  Serial.print(now); Serial.print(F(" ms  leg ")); Serial.print(leg + 1); Serial.print(F
  Serial.println(failed ? F("  FAILED: a leg is out of reach or out of the servos' range
}

void setup() {
  Serial.begin(115200);
  s_shoulder.attach(PIN_SHOULDER); s_elbow.attach(PIN_ELBOW); s_grip.attach(PIN_GRIP);
  s_grip.write(GRIP_OPEN);
  float t1, t2;
  ik_lut(ARM, 110.0f, 40.0f, true, t1, t2, ACOS_T, ATAN_T, LUT_N);
  cur1 = t1 * 57.29578f; cur2 = t2 * 57.29578f; write_joints(cur1, cur2);
  delay(1000);                                                     // let the arm reach
  running = start_leg(0, millis()); failed = !running;
  sch.add(control_task, 0, 20);
  sch.add(report_task, 0, 500);
}

void loop() { sch.run(millis()); }
```

It compiles for the Uno under `--warnings all` with no warning from the sketch to:

```
Sketch uses 10040 bytes (31%) of program storage space. Maximum is 32256 bytes.
Global variables use 437 bytes (21%) of dynamic memory, leaving 1611 bytes for local var
```

That is 10,040 bytes of flash and 437 bytes of RAM. It was **not run**: there is no arm, the link lengths, zero positions, gripper angles and limits are guesses, and I never saw a servo move. The first thing to do on a real arm is to measure the two link lengths and the real range of each servo, then to move one joint at a time with `writeMicroseconds` at a small speed.

8. What the numbers can and cannot be trusted for

What they show. The shapes: the inverse kinematics is exact and refuses what it cannot reach; the tip error from a servo step is about the reach times the step, so whole degrees give millimetres and microseconds give a tenth of a millimetre; a trapezoid profile is exact and a synchronised pair of joints arrives together; whole-degree commands turn a slow move into stop-and-go; a lookup table is accurate in the middle of its range and poor at the ends where the function is steep; joint interpolation curves the tip's path by several millimetres; the sizes of every variant on the Uno and the counts of instructions and library calls.

What they do not show. The arm is a simulation with link lengths, limits and play that I chose: a real servo has a speed limit and a torque limit that depend on its load and voltage, a backlash that is not a symmetric deadband, a pulse-to-angle map that differs from the library's 544–2400, and it does not follow a command in a straight ramp. The Servo-library behaviour is read from source, not seen on a pin. No gravity, no payload, no gripper geometry, no collision except with the table. **No time was measured**: the Uno figures are bytes, instructions and calls, so "the table is faster" and "integers are faster" are expectations, not results. `robot_ch11.ino` was compiled, not run.

CHALLENGE 1: THE SERVO STEP FOR HALF A MILLIMETRE

Round both joint angles to steps from 2° down to 0.025° and measure the tip error over the working area ($y \geq -40$ mm, $30 \leq r \leq 170$ mm), overall and in three rings of the distance from the shoulder. Which is the coarsest step that places the tip within 0.5 mm everywhere, what is it in microseconds, and in which ring is the error worst?

View solution

CHALLENGE 2: TWO TABLES OF DIFFERENT SIZES

Give `ik_lut()` an acos table and an atan table of different sizes (8, 16, 32, 64 and 128 entries each) and report the worst tip error over 40 ≤ r ≤ 170 mm for every pair. Which table decides the accuracy, what is the cheapest pair in flash that stays below 0.5 mm, and what does it save against two tables of 64?

View solution

CHALLENGE 3: A STRAIGHT LINE AGAINST A CURVE

For three legs of the cycle compare the joint-space move with a Cartesian move (a trapezoid along the straight line, inverse kinematics at every 20 ms step): how far the tip strays from the straight line, and how long the leg takes when the Cartesian move is run at the largest tip speed at which no joint exceeds 120°/s or 400°/s². When is the straight line worth the extra time?

View solution

QUICK REFERENCE

- **Kinematics.** `x = l1 cos t1 + l2 cos(t1+t2)`, `cos t2 = (x²+y²−l1²−l2²)/(2 l1 l2)`; outside [−1, 1] return `false`. Round trip within 0.00006 mm over 100,520 points; reachable ring r = 20–180 mm for 100 and 80 mm links.
- **Servo step.** Tip error is about reach × step: whole degrees 0.63 mm mean, 2.07 worst; 1 µs (0.097°) 0.06 and 0.20; 0.5 µs tick 0.03 and 0.10. 0.1° meets 0.5 mm, 0.25° does not (0.52 mm).
- **The edge.** Moving the tip outward at 50 mm/s takes 12–58°/s between 40 and 150 mm and 637 and 1,433°/s at 179.95 mm: keep waypoints away from the straight arm.
- **Servo library (read, not run).** 544–2400 µs for 0–180°; `write(int)` is whole degrees; Timer1 at 0.5 µs per tick; 20 ms frame, servos pulsed one after another, 12 per timer; `attach` puts Timer1 in normal mode, so I expect `analogWrite` on pins 9 and 10 to stop (not run).
- **Trapezoid.** Triangle if `d < vmax²/amax` (36° for 120 and 400); synchronise by scaling the shorter joint's limits by the ratio of distances: 60° and 15° both finish at 0.800 s. Whole-degree commands at a slow speed give stop-and-go: 47 of 77 updates still, 1° jumps at 50°/s for a 20°/s move.
- **Lookup tables.** n = 64: 0.16 mm worst in 40–170 mm, about 5 mm at both ends (acos is steep at ±1). They did *not* save flash: ik +2122 bytes, ik_lut +2592 (260 bytes of tables); speed not measured. The integer sine table: fk_int +588 against +1452 for the float fk.

- **Pick and place (simulated, play 0.8°).** 3.9 s of commanded motion per cycle; tip error 0.70 mm with microsecond steps (play is the floor), 0.94 mm mean and 2.1 mm worst with whole degrees; a joint-space "straight down" drifts 5–8 mm sideways.
- **Budget.** `robot_ch11.ino`: 10,040 bytes flash (31 %), 437 bytes RAM (21 %), compiled, not run. Two Servo objects cost +1632 flash and +44 RAM; three servos use pins 9, 10 and 13 (the LED).
- **Not run:** anything on a board; any real arm or servo; the speed of a table against a library routine, or of integers against floats.

Next: Programming for Robotics with Arduino C++ 12, Testing, Safety and Where to Go Next

Programming for Robotics with Arduino C++ 12 (planned; not yet written) closes the course: testing robot code on the PC before it touches hardware, the safety rules a robot that moves must follow (limits, watchdogs, an emergency stop that does not depend on the code that failed), a review of what the course measured and what it only compiled, and where to go next.

Programming for Robotics with Arduino C++

Course 2 · Chapter 12 · Testing, Safety and Where to Go Next

Eleven chapters have built a robot that senses, filters, controls, plans, estimates and reaches. This last chapter is about the question that sits under all of it: *how do you know it works, and what does it do when it does not?* A robot is code that moves things in the world, so a bug is not a wrong number on a screen but a wall that is hit or a motor that burns. The honest answer to the first question is a test suite that you have seen fail on purpose. The honest answer to the second is a set of rules that make **stopped** the state every failure falls into, built from pieces that do not depend on the code they are guarding.

The chapter has five parts: unit tests for the host-built logic of the whole course, *mutation testing* to find out whether those tests would notice a bug, *fault injection* on the simulator to find out whether the safety pieces really stop the robot, the same pieces on the Uno with their cost, and a short look at what lies beyond an 8-bit board. It ends with a review of what this course measured and what it only compiled.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Run on a PC** (g++ 16.2, C++11, `-Wall -Wextra -pedantic`): the unit tests, the mutation runs, the fault-injection runs and the three challenge solutions, against the simulated robot of Programming for Robotics with Arduino C++ 3. The faults, the late controller, the sensor noise and the arena are **assumptions I chose**. **Compiled for the Uno, not run** (arduino-cli, core `arduino:avr` 1.8.8, `--warnings all`): the seven-variant cost-test sketch and `robot_ch12.ino`; instruction counts come from `avr-objdump`. The watchdog, the emergency-stop input and the ring-buffer dump were **never run on a board**. **Read from the core's own files:** the Uno's fuse bytes (`boards.txt`). **From memory and not checked:** what those bytes mean (the brown-out level) and everything said about faster boards, an RTOS and micro-ROS. **Nothing was attached to a board and no time was measured.** A test that passes on the PC shows the logic is right on the PC; it does not show the board behaves.

1. A test suite in twenty lines of harness

Because the course wrote its logic as plain C++ with no Arduino calls, it can be tested on the PC without a board, a framework or a download. The whole test harness is a macro that counts passes and failures and says which line failed:

```
// minitest.h -- a test harness in twenty lines (host only). No framework: a macro that
#pragma once
#include <cstdio>
#include <cmath>

static int mt_pass = 0, mt_fail = 0;
#define CHECK(cond) do { if (cond) ++mt_pass; else { ++mt_fail; std::printf("  FAIL line
#define CHECK_NEAR(a, b, tol) do { double _a = (a), _b = (b); if (std::fabs(_a - _b) <=
#define TEST(name) static void name()
#define RUN(name) do { int f0 = mt_fail; name(); std::printf("  %-34s %s\n", #name, mt_f
static int mt_summary() { std::printf("%d checks passed, %d failed\n", mt_pass, mt_fail)
```

The test file covers the pieces of the whole course: time and the scheduler (including the 32-bit counter wrap), the PID anti-windup, the planner, inverse kinematics and its lookup table, the trapezoid profile, the three estimators, and the new safety pieces of this chapter. Each test states one thing with a number that comes from the maths or from a definition, not copied from what the code printed: a trapezoid covers its distance; a BFS round a wall with a gap at the top is 4 + 7 + 3 + 7 = 21 steps; a scalar Kalman filter's gain settles at `P / (P + R)` with the steady-state `P` of the formula; a complementary filter fed a constant angle converges to it.

```
// tests.cpp -- unit tests of the host-built logic of the whole course (Programming for
// Each test states one thing the code must do, with a number that comes from the maths
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <initializer_list>
#include "minitest.h"
#include "timing.h"
#include "scheduler.h"
#include "pid.h"
#include "grid.h"
#include "tables.h"
#include "arm.h"
#include "est.h"
#include "safety.h"
#include "ringlog.h"

// ---- timing and the scheduler ----
static int g_calls = 0;
static void count_task(void*, uint32_t) { ++g_calls; }
```

```
TEST(elapsed_survives_the_wrap) {
  CHECK(elapsed(5u, 0xFFFFFFFBu) == 10u);                                  // the counter wr
  CHECK(elapsed(100u, 40u) == 60u);
}
TEST(periodic_runs_once_per_period) {
  Periodic p(10); int runs = 0;
  for (uint32_t t = 0; t < 1000; ++t) if (p.due(t)) ++runs;
  CHECK(runs == 100);                                                      // at 0, 10, ...,
}
TEST(periodic_survives_the_wrap) {
  Periodic p(10); int runs = 0;
  uint32_t t0 = 0xFFFFFF00u;
  for (uint32_t i = 0; i < 1000; ++i) if (p.due(t0 + i)) ++runs;      // 1000 ms startin
  CHECK(runs == 100);
}
TEST(scheduler_runs_tasks_at_their_periods) {
  Scheduler<2> s; g_calls = 0;
  s.add(count_task, 0, 20);
  for (uint32_t t = 0; t < 1000; ++t) s.run(t);
  CHECK(g_calls == 50);
}

// ---- control ----
TEST(pid_anti_windup_bounds_the_integral) {
  PidConfig c = {1.0f, 2.0f, 0.0f, -1.0f, 1.0f, 0.0f, true, AW_CLAMP, 0.0f};
  PidConfig n = c; n.aw = AW_NONE;
  Pid a(c), b(n);
  for (int i = 0; i < 500; ++i) { a.update(10.0f, 0.0f, 0.01f); b.update(10.0f, 0.0f, 0.
  CHECK(a.integral() <= 1.0f + 1e-3f);                                     // clamped: the i
  CHECK(b.integral() > 50.0f);                                             // unclamped: 2 x
}

// ---- planning ----
TEST(bfs_goes_left_too) {
  BitGrid<8, 8> g; Bfs<8, 8, 64> bfs; uint16_t path[64];
  CHECK(bfs.plan(g, 7, 3, 0, 3, path, 64) == 7);                           // an open grid,
  CHECK(bfs.plan(g, 3, 7, 3, 0, path, 64) == 7);                           // and upward
}
TEST(bfs_finds_the_shortest_path_round_a_wall) {
  BitGrid<8, 8> g;
  for (uint8_t y = 0; y < 7; ++y) g.set(4, y, true);                       // a wall at x =
  Bfs<8, 8, 64> bfs; uint16_t path[64];
  int16_t n = bfs.plan(g, 0, 0, 7, 0, path, 64);
  CHECK(n == 21);                                                          // 4 + 7 to the g
  CHECK(bfs.plan(g, 0, 0, 4, 3, path, 64) == -1);                          // the goal is in
```

```
  for (uint8_t y = 0; y < 8; ++y) g.set(4, y, true);
  CHECK(bfs.plan(g, 0, 0, 7, 0, path, 64) == -1);                       // no gap at all:
}

// ---- kinematics and profiles ----
TEST(atan2_table_matches_atan2_in_all_quadrants) {
  const float pts[9][2] = {{1, 0}, {1, 1}, {0, 1}, {-1, 1}, {-1, 0}, {-1, -1}, {0, -1}, 
  for (int i = 0; i < 9; ++i) CHECK_NEAR(atan2_lut(pts[i][1], pts[i][0], ATAN_T, LUT_N),
}
TEST(ik_round_trip_and_reach) {
  Arm a = {100.0f, 80.0f}; float t1 = 0, t2 = 0, x, y;
  for (int px = -170; px <= 170; px += 17)
    for (int py = -170; py <= 170; py += 17) {
      bool ok = ik(a, (float)px, (float)py, true, t1, t2);
      double r = std::hypot((double)px, (double)py);
      if (r > 21.0 && r < 179.0) { CHECK(ok); if (ok) { fk(a, t1, t2, x, y); CHECK_NEAR(
      if (r > 181.0 || r < 19.0) CHECK(!ok);
    }
}
TEST(ik_elbow_sign) {
  Arm a = {100.0f, 80.0f}; float u1 = 0, u2 = 0, d1 = 0, d2 = 0;
  CHECK(ik(a, 120.0f, 40.0f, true, u1, u2));
  CHECK(ik(a, 120.0f, 40.0f, false, d1, d2));
  CHECK(u2 < 0.0f && d2 > 0.0f);                                        // up: the elbow 
}
TEST(lut_ik_is_close_to_the_math_ik) {
  Arm a = {100.0f, 80.0f}; float s1 = 0, s2 = 0, l1 = 0, l2 = 0;
  for (int px = 60; px <= 140; px += 20)
    for (int py = -30; py <= 60; py += 15) {
      CHECK(ik(a, (float)px, (float)py, true, s1, s2));
      CHECK(ik_lut(a, (float)px, (float)py, true, l1, l2, ACOS_T, ATAN_T, LUT_N));
      float x, y; fk(a, l1, l2, x, y);
      CHECK(std::hypot(x - px, y - py) < 0.5);                          // half a millime
    }
}
TEST(integer_fk_matches_float_fk) {
  Arm a = {100.0f, 80.0f};
  for (int t1 = 0; t1 <= 90; t1 += 15)
    for (int t2 = -150; t2 <= 30; t2 += 30) {
      float x, y; fk(a, t1 * 0.0174532925f, t2 * 0.0174532925f, x, y);
      int16_t xi, yi; fk_int(100, 80, (int16_t)t1, (int16_t)t2, xi, yi);
      CHECK(std::fabs(xi - x * 10.0f) < 3.0f && std::fabs(yi - y * 10.0f) < 3.0f);   // 
    }
}
TEST(trapezoid_covers_the_distance_in_the_planned_time) {
```

```
  for (float d : {5.0f, 20.0f, 30.0f, 36.0f, 90.0f, -60.0f}) {
    Trap t; t.plan(d, 120.0f, 400.0f);
    CHECK_NEAR(t.pos(t.T), d, 1e-4);
    CHECK_NEAR(t.pos(t.T * 0.5f), d * 0.5f, std::fabs(d) * 1e-3);       // symmetric: ha
    CHECK(t.vel(0.0f) == 0.0f && t.vel(t.T) == 0.0f);
    double integ = 0; for (int i = 1; i <= (int)(t.T * 1000); ++i) integ += t.vel(i * 0.
    CHECK_NEAR(integ, d, std::fabs(d) * 0.01);
  }
  Trap tri; tri.plan(10.0f, 120.0f, 400.0f);
  CHECK_NEAR(tri.v, std::sqrt(400.0 * 10.0), 1e-3);                     // a short move i
  Trap a, b; plan_sync(a, b, 60.0f, 15.0f, 120.0f, 400.0f);
  CHECK_NEAR(a.T, b.T, 1e-4);                                           // two joints fin
}

// ---- estimation ----
TEST(complementary_filter_believes_the_gyro_in_the_short_term) {
  Complementary f(0.5f, 0.01f, 0.0f);
  for (int i = 0; i < 10; ++i) f.update(1.0f, 0.0f, 0.01f);             // a gyro saying
  CHECK(f.angle() > 0.08f && f.angle() < 0.1f);                         // the gyro wins
}
TEST(complementary_filter_converges_to_a_constant_angle) {
  Complementary f(0.5f, 0.01f, 0.0f);
  for (int i = 0; i < 2000; ++i) f.update(0.0f, 0.3f, 0.01f);           // 20 s at 40 tim
  CHECK_NEAR(f.angle(), 0.3, 1e-3);
}
TEST(scalar_kalman_gain_reaches_the_formula) {
  const float q = 1e-6f, r = 2.5e-5f;
  Kalman1 k(0.0f, 1.0f, q, r);
  for (int i = 0; i < 500; ++i) k.update(1.0f);
  double p = (q + std::sqrt((double)q * q + 4.0 * q * r)) / 2.0;        // steady-state p
  CHECK_NEAR(k.gain(), p / (p + r), 1e-3);
  CHECK_NEAR(k.x(), 1.0, 1e-3);
}
TEST(fixed_point_heading_filter_learns_the_gyro_offset) {
  HeadingFixed f(1223, -114);
  for (int i = 1; i <= 6000; ++i) {                                     // a still robot,
    f.predict(20000);
    if (i % 10 == 0) f.update(0);
  }
  CHECK(std::abs(f.heading()) < 20000);                                // within 0.02 rad
  CHECK(std::abs(f.bias() - 20000) < 4000);                            // the offset is f
}

// ---- the safety pieces ----
TEST(output_expires) {
```

```
  ExpiringCmd e(100); Cmd c = {0.3f, 0.3f};
  CHECK(e.expired(0));                                          // nothing given y
  e.set(c, 1000);
  CHECK(e.get(1050).vl == 0.3f);
  CHECK(e.get(1100).vl == 0.3f);                                // exactly at the
  CHECK(e.get(1101).vl == 0.0f && e.get(1101).vr == 0.0f);
  e.set(c, 0xFFFFFFF0u);
  CHECK(e.get(20).vl == 0.3f);                                  // 36 ms later, a
}
TEST(estop_is_debounced_and_latched) {
  Estop s(3);
  s.sample(true); s.sample(false); s.sample(true); s.sample(true);
  CHECK(!s.tripped());                                          // bounce: never t
  s.sample(true);
  CHECK(s.tripped());
  s.sample(false);
  CHECK(s.tripped());                                           // releasing the l
  s.sample(true);
  CHECK(!s.reset());                                            // reset is refus
  s.sample(false);
  CHECK(s.reset() && !s.tripped());
}
TEST(stuck_detector_needs_motion_and_no_change) {
  StuckDetector d(0.02f, 500);
  for (uint32_t t = 0; t <= 1000; t += 60) d.update(1.0f, true, false, t);   // still: a
  CHECK(!d.stuck(1000));
  StuckDetector m(0.02f, 500);
  for (uint32_t t = 0; t <= 400; t += 60) m.update(1.0f, true, true, t);
  CHECK(!m.stuck(400));                                         // not yet: the wi
  for (uint32_t t = 420; t <= 800; t += 60) m.update(1.0f, true, true, t);
  CHECK(m.stuck(800));
  StuckDetector ok(0.02f, 500); float r = 2.0f;
  for (uint32_t t = 0; t <= 2000; t += 60) { ok.update(r, true, true, t); r -= 0.015f; }
  CHECK(!ok.stuck(2000));                                       // a reading that
  StuckDetector slow(0.02f, 500); float q = 2.0f;
  for (uint32_t t = 0; t <= 2000; t += 60) { slow.update(q, true, true, t); q -= 0.0036f;
  CHECK(!slow.stuck(2000));
}
TEST(stall_detector_sees_a_blocked_wheel) {
  StallDetector s(0.05f, 400);
  for (uint32_t t = 0; t <= 300; t += 20) s.update(0.3f, 0, t);
  CHECK(!s.stalled(300));
  for (uint32_t t = 320; t <= 500; t += 20) s.update(0.3f, 0, t);
  CHECK(s.stalled(500));
  StallDetector w(0.05f, 400); int32_t tk = 0;
```

```
  for (uint32_t t = 0; t <= 2000; t += 20) { tk += 3; w.update(0.3f, tk, t); }
  CHECK(!w.stalled(2000));                                          // turning: no st
}
TEST(supervisor_stops_what_it_must_and_passes_the_rest) {
  SafetyConfig c = {250, 300, 0.02f, 500, 400, 0.15f, 0.4f};
  Supervisor s(c); Cmd go = {0.3f, 0.3f};
  s.command(go, 0);
  int32_t tk = 0; Cmd out = go;
  for (uint32_t t = 20; t <= 200; t += 20) { tk += 6; s.range_sample(t, 2.0f - 0.001f *
  CHECK(out.vl == 0.3f && s.faults() == 0);                         // healthy: passe
  out = s.step(200 + 400, tk, tk);
  CHECK(out.vl == 0.0f && out.vr == 0.0f);                          // the command is
  s.command(go, 700);
  s.range_sample(710, 0.10f);
  out = s.step(720, tk + 6, tk + 6);
  CHECK(out.vl == 0.0f && (s.faults() & F_TOO_CLOSE));              // forward with a
  s.command(go, 740);
  s.range_sample(750, 1.5f);
  out = s.step(760, tk + 12, tk + 12);
  CHECK(out.vl == 0.0f);                                            // still latched
  CHECK(s.reset() && s.faults() == 0);
  Cmd fast = {2.0f, -2.0f}; s.command(fast, 800);
  s.range_sample(810, 3.0f);
  out = s.step(820, tk + 20, tk + 20);
  CHECK(out.vl == 0.4f && out.vr == -0.4f);                         // the speed limi
  Supervisor st(c); st.command(go, 1000);
  for (uint32_t t = 1020; t <= 1400; t += 20) { st.command(go, t); st.range_sample(t, -1
  CHECK(st.faults() & F_STALE_RANGE);                               // no valid readi
}

// ---- the log ----
TEST(ring_log_keeps_the_last_events) {
  RingLog<4> log;
  for (int i = 1; i <= 10; ++i) log.add((uint32_t)i * 10, (uint8_t)i, (int16_t)(i * 100)
  CHECK(log.size() == 4 && log.total() == 10);
  CHECK(log.at(0).code == 7 && log.at(3).code == 10);               // the oldest kep
  CHECK(log.at(0).value == 700 && log.at(3).t_ms == 100);
  RingLog<4> few; few.add(1, 1, 1); few.add(2, 2, 2);
  CHECK(few.size() == 2 && few.at(0).code == 1 && few.at(1).code == 2);   // not yet ful
}

int main() {
  std::printf("unit tests of the host-built logic\n");
  RUN(elapsed_survives_the_wrap);
  RUN(periodic_runs_once_per_period);
```

```
  RUN(periodic_survives_the_wrap);
  RUN(scheduler_runs_tasks_at_their_periods);
  RUN(pid_anti_windup_bounds_the_integral);
  RUN(bfs_goes_left_too);
  RUN(bfs_finds_the_shortest_path_round_a_wall);
  RUN(atan2_table_matches_atan2_in_all_quadrants);
  RUN(ik_round_trip_and_reach);
  RUN(ik_elbow_sign);
  RUN(lut_ik_is_close_to_the_math_ik);
  RUN(integer_fk_matches_float_fk);
  RUN(trapezoid_covers_the_distance_in_the_planned_time);
  RUN(complementary_filter_believes_the_gyro_in_the_short_term);
  RUN(complementary_filter_converges_to_a_constant_angle);
  RUN(scalar_kalman_gain_reaches_the_formula);
  RUN(fixed_point_heading_filter_learns_the_gyro_offset);
  RUN(output_expires);
  RUN(estop_is_debounced_and_latched);
  RUN(stuck_detector_needs_motion_and_no_change);
  RUN(stall_detector_sees_a_blocked_wheel);
  RUN(supervisor_stops_what_it_must_and_passes_the_rest);
  RUN(ring_log_keeps_the_last_events);
  return mt_summary();
}
```

```
unit tests of the host-built logic
  elapsed_survives_the_wrap          ok
  periodic_runs_once_per_period      ok
  periodic_survives_the_wrap         ok
  scheduler_runs_tasks_at_their_periods ok
  pid_anti_windup_bounds_the_integral ok
  bfs_goes_left_too                  ok
  bfs_finds_the_shortest_path_round_a_wall ok
  atan2_table_matches_atan2_in_all_quadrants ok
  ik_round_trip_and_reach            ok
  ik_elbow_sign                      ok
  lut_ik_is_close_to_the_math_ik     ok
  integer_fk_matches_float_fk        ok
  trapezoid_covers_the_distance_in_the_planned_time ok
  complementary_filter_believes_the_gyro_in_the_short_term ok
  complementary_filter_converges_to_a_constant_angle ok
  scalar_kalman_gain_reaches_the_formula ok
  fixed_point_heading_filter_learns_the_gyro_offset ok
  output_expires                     ok
  estop_is_debounced_and_latched     ok
```

```
  stuck_detector_needs_motion_and_no_change ok
  stall_detector_sees_a_blocked_wheel ok
  supervisor_stops_what_it_must_and_passes_the_rest ok
  ring_log_keeps_the_last_events     ok
1360 checks passed, 0 failed
```

2. Would the tests notice a bug?

The first version of this suite had 1,337 checks, and it passed the first time it was run. That is the dangerous moment: a green suite proves nothing if it cannot go red. **Mutation testing** answers the question mechanically. Take the code, change *one thing* that a careless edit could change (a sign, a `>` for `>=`, a forgotten term), build the tests against the changed code and run them. If a check fails, the mutant is *killed*: the suite would have caught that bug. If everything still passes, the mutant *survived*: the suite is blind to it. The script below does that for 21 mutants:

```
# mutate.py -- does the test suite notice a bug? Each mutant is a copy of the code with
# "killed" = at least one check failed (or the program crashed or hung); "survived" = ev
import os, re, shutil, subprocess, sys

HERE = os.path.dirname(os.path.abspath(__file__))
GPP = r"C:/Users/emuba/AppData/Local/Microsoft/WinGet/Packages/BrechtSanders.WinLibs.POS
HEADERS = [f for f in os.listdir(HERE) if f.endswith(".h")] + ["tests.cpp"]

# (file, text that must occur, replacement, what the mutant is)
M = [
    ("arm.h", "if (elbow_up) s2 = -s2;", "if (!elbow_up) s2 = -s2;", "ik: the elbow-up a
    ("arm.h", "if (c2 > 1.0f || c2 < -1.0f) return false;", "if (c2 > 1.0f) return false
    ("arm.h", "T = 2.0f * ta + tc;", "T = ta + tc;", "Trap: the braking phase left out o
    ("arm.h", "if (d < vmax * vmax / amax)", "if (d < 0.5f * vmax * vmax / amax)", "Trap
    ("arm.h", "if (ax == 0.0f && ay == 0.0f) return 0.0f;", "if (ax == 0.0f && ay == 0.0
    ("est.h", "alpha_(tau_s / (tau_s + dt_s))", "alpha_(dt_s / (tau_s + dt_s))", "Comple
    ("est.h", "p_ *= (1.0f - k_);", "p_ *= k_;", "Kalman1: the uncertainty update uses K
    ("est.h", "th_ += (rate_urad_s - b_) / 100;", "th_ += (rate_urad_s + b_) / 100;", "H
    ("timing.h", "inline uint32_t elapsed(uint32_t now, uint32_t then) { return now - th
    ("pid.h", "u > c_.out_max && e > 0.0f", "u > c_.out_max && e < 0.0f", "Pid: the anti
    ("grid.h", "if (m.blocked(sx, sy) || m.blocked(gx, gy)) return -1;", "if (m.blocked(
    ("grid.h", "inline int8_t dx4(uint8_t d) { return d == 0 ? 1 : (d == 2 ? -1 : 0); }"
    ("safety.h", "if (count_ >= n_) latched_ = true;", "if (count_ > n_) latched_ = true
    ("safety.h", "if (released_) { latched_ = false; return true; } return false;", "lat
    ("safety.h", "elapsed(now, stamp_) > timeout_", "elapsed(now, stamp_) >= timeout_",
    ("safety.h", "if (c < min_cmd_ || !have_ || ticks != last_ticks_)", "if (c < min_cmd_
```

```
    ("safety.h", "else if (out.vr < -c_.v_max) out.vr = -c_.v_max;", "else if (out.vr <
    ("safety.h", "if (faults_ != 0) { out.vl = 0.0f; out.vr = 0.0f; }", "if (faults_ !=
    ("safety.h", "if (d > delta_) { anchor_ = reading_m; since_ = now; }", "if (d > 2.0f
    ("ringlog.h", "head_ = (uint8_t)((head_ + 1) % N);", "head_ = (uint8_t)((head_ + 1)
    ("ringlog.h", "if (count_ < N) ++count_;", "if (count_ < N - 1) ++count_;", "RingLog
]

def run(cmd, **kw):
    return subprocess.run(cmd, capture_output=True, text=True, **kw)

killed = survived = invalid = 0
print("%-3s %-62s %s" % ("#", "mutant (one change)", "result"))
for i, (fname, old, new, what) in enumerate(M, 1):
    d = os.path.join(HERE, "mut", "m%02d" % i)
    shutil.rmtree(d, ignore_errors=True); os.makedirs(d)
    for h in HEADERS:
        shutil.copy(os.path.join(HERE, h), d)
    p = os.path.join(d, fname)
    s = open(p, encoding="utf-8").read()
    if old not in s:
        print("%-3d %-62s NOT APPLIED (text not found)" % (i, what)); invalid += 1; cont
    open(p, "w", encoding="utf-8", newline="\n").write(s.replace(old, new, 1))
    exe = os.path.join(d, "t.exe")
    c = run([GPP, "-std=c++11", "-O1", os.path.join(d, "tests.cpp"), "-o", exe], cwd=d)
    if c.returncode != 0:
        print("%-3d %-62s DID NOT COMPILE" % (i, what)); invalid += 1; continue
    try:
        r = run([exe], cwd=d, timeout=20)
        out = r.stdout
        m = re.search(r"(\d+) checks passed, (\d+) failed", out)
        if r.returncode != 0 and m:
            failed_tests = [l.split()[0] for l in out.splitlines() if l.rstrip().endswit
            print("%-3d %-62s killed: %s checks failed (%s)" % (i, what, m.group(2), "
        elif r.returncode != 0:
            print("%-3d %-62s killed: the program crashed" % (i, what)); killed += 1
        else:
            print("%-3d %-62s SURVIVED" % (i, what)); survived += 1
    except subprocess.TimeoutExpired:
        print("%-3d %-62s killed: the program hung (20 s)" % (i, what)); killed += 1
print("%d mutants: %d killed, %d survived, %d invalid" % (len(M), killed, survived, inva
```

First run, against that first suite:

```
  #   mutant (one change)                                    result
  1   ik: the elbow-up and elbow-down branches swapped       killed: 1 checks fail
  2   ik: the too-close side of the reach test removed       killed: 5 checks fail
  3   Trap: the braking phase left out of the total time     killed: 8 checks fail
  4   Trap: the triangle/trapezoid threshold halved          SURVIVED
  5   atan2_lut: wrong answer for the point at the origin    SURVIVED
  6   Complementary: gyro and measurement weights swapped    SURVIVED
  7   Kalman1: the uncertainty update uses K instead of 1 - K  killed: 1 checks fail
  8   HeadingFixed: the offset is added instead of subtracted  killed: 2 checks fail
  9   elapsed: the counter wrap is no longer handled         killed: 2 checks fail
  10  Pid: the anti-windup test looks at the wrong sign      killed: 1 checks fail
  11  Bfs: a blocked goal is no longer rejected up front     SURVIVED
  12  Bfs: the 'left' neighbour is replaced by 'right'       SURVIVED
  13  Estop: debounce needs one sample more than asked       killed: 2 checks fail
  14  Estop: reset accepted while the button is still held   killed: 1 checks fail
  15  ExpiringCmd: expires at the timeout instead of after it  SURVIVED
  16  StallDetector: wheel movement no longer resets the timer  killed: 1 checks fail
  17  Supervisor: the reverse speed limit has the wrong sign  killed: 1 checks fail
  18  Supervisor: a fault no longer stops the motors         killed: 2 checks fail
  19  StuckDetector: the change threshold doubled            SURVIVED
  20  RingLog: the write position wraps one slot early       killed: 2 checks fail
  21  RingLog: one slot of the buffer is never counted       killed: 3 checks fail
  21 mutants: 14 killed, 7 survived, 0 invalid
```

Seven of 21 survived, and each survivor was a hole in the tests, not in the code. The trapezoid's triangle threshold was halved and nothing noticed, because no test used a move between 18° and 36°. The complementary filter's two weights were *swapped* and nothing noticed, because the only test fed it a constant angle, which the swapped filter reaches even faster; it never tested that the gyro is believed in the short term. The breadth-first search lost its "left" neighbour and nothing noticed, because every test path went up and to the right. The command timeout fired at the timeout instead of after it, because no test sat exactly on the boundary. The stuck detector's threshold was doubled, because no test had a reading that drifted slowly. The point at the origin of the table-based arctangent was wrong, because no test asked for it.

I added tests to close those holes (a trapezoid of 20° and 30°; the gyro in the short term; a path to the left and downward; a command at exactly its timeout; a slow drift; the arctangent table in all quadrants) and ran it again:

```
  #   mutant (one change)                                    result
  1   ik: the elbow-up and elbow-down branches swapped       killed: 1 checks fail
  2   ik: the too-close side of the reach test removed       killed: 5 checks fail
```

```
  3   Trap: the braking phase left out of the total time            killed: 12 checks fai
  4   Trap: the triangle/trapezoid threshold halved                 killed: 4 checks fail
  5   atan2_lut: wrong answer for the point at the origin           killed: 1 checks fail
  6   Complementary: gyro and measurement weights swapped           killed: 1 checks fail
  7   Kalman1: the uncertainty update uses K instead of 1 - K       killed: 1 checks fail
  8   HeadingFixed: the offset is added instead of subtracted       killed: 2 checks fail
  9   elapsed: the counter wrap is no longer handled                killed: 2 checks fail
  10  Pid: the anti-windup test looks at the wrong sign             killed: 1 checks fail
  11  Bfs: a blocked goal is no longer rejected up front            SURVIVED
  12  Bfs: the 'left' neighbour is replaced by 'right'              killed: 1 checks fail
  13  Estop: debounce needs one sample more than asked              killed: 2 checks fail
  14  Estop: reset accepted while the button is still held          killed: 1 checks fail
  15  ExpiringCmd: expires at the timeout instead of after it       killed: 1 checks fail
  16  StallDetector: wheel movement no longer resets the timer      killed: 1 checks fail
  17  Supervisor: the reverse speed limit has the wrong sign        killed: 1 checks fail
  18  Supervisor: a fault no longer stops the motors                killed: 2 checks fail
  19  StuckDetector: the change threshold doubled                   killed: 1 checks fail
  20  RingLog: the write position wraps one slot early              killed: 2 checks fail
  21  RingLog: one slot of the buffer is never counted              killed: 3 checks fail
  21 mutants: 20 killed, 1 survived, 0 invalid
```

Twenty of 21 are killed (the suite now has 1,360 checks). The survivor, a blocked goal no longer rejected up front in `Bfs`, is an *equivalent mutant*: the search never enqueues a blocked cell, so a blocked goal returns "unreachable" either way and no test can tell the two versions apart. Equivalent mutants are the one kind of survivor you stop worrying about, once you have checked that it really is one. Mutation testing has limits: 21 hand-picked mutants are a sample, not a proof, and the tests only check what the host build can reach (nothing here tests the interrupts or the pins).

3. Safety is a design, not a feature

Three principles run through the code of this chapter. **The safe state is stopped**, and every part fails toward it. **The guard must not depend on the thing it guards**: a supervisor that stops working when the controller hangs is no supervisor. **A fault is latched**: when something has gone wrong the robot stays stopped until a person has looked and reset it, not until the condition happens to go away. `safety.h` turns them into five small pieces, plain C++ with no heap and no Arduino calls so the PC can fault-inject them:

```
// safety.h -- the rules that keep a moving robot from doing damage, as small testable p
// Plain C++11: no heap, no Arduino calls, so a PC can test and fault-inject it. Times a
```

```
// The principle: the safe state is "stopped". Every piece fails toward it, and nothing
#pragma once
#include <stdint.h>
#include "timing.h"
#include "fsm.h"                                    // Cmd { vl, vr }

enum Fault : uint8_t {
  F_NONE = 0, F_ESTOP = 1, F_STALE_CMD = 2, F_STALE_RANGE = 4, F_STUCK_RANGE = 8, F_STAL
};

// ---- 1. an output that expires: a command is only good for `timeout_ms` after it was
// If the controller hangs, crashes or is late, the motors do not keep the last command:
class ExpiringCmd {
 public:
  explicit ExpiringCmd(uint32_t timeout_ms) : timeout_(timeout_ms), stamp_(0), have_(fal
  void set(const Cmd& c, uint32_t now) { cmd_ = c; stamp_ = now; have_ = true; }
  bool expired(uint32_t now) const { return !have_ || elapsed(now, stamp_) > timeout_; }
  Cmd get(uint32_t now) const { Cmd z = {0.0f, 0.0f}; return expired(now) ? z : cmd_; }
 private:
  uint32_t timeout_, stamp_;
  bool have_;
  Cmd cmd_;
};

// ---- 2. an emergency-stop input: debounced and LATCHED ----
// The switch must read "pressed" for `n` consecutive samples to trip; once tripped the
// a bounce or a released button must not restart the robot.
class Estop {
 public:
  explicit Estop(uint8_t n) : n_(n), count_(0), latched_(false), released_(true) {}
  void sample(bool pressed) {
    if (pressed) { if (count_ < 255) ++count_; released_ = false; if (count_ >= n_) latc
    else { count_ = 0; released_ = true; }
  }
  bool reset() { if (released_) { latched_ = false; return true; } return false; }
  bool tripped() const { return latched_; }
 private:
  uint8_t n_, count_;
  bool latched_, released_;
};

// ---- 3. a stuck-sensor detector: while the robot is moving, a distance reading that d
class StuckDetector {
 public:
  StuckDetector(float delta, uint32_t window_ms) : delta_(delta), window_(window_ms), an
```

```
  void update(float reading_m, bool valid, bool moving, uint32_t now) {
    if (!moving || !valid) { have_ = false; return; }               // restart the windo
    if (!have_) { anchor_ = reading_m; since_ = now; have_ = true; return; }
    float d = reading_m - anchor_; if (d < 0.0f) d = -d;
    if (d > delta_) { anchor_ = reading_m; since_ = now; }
  }
  bool stuck(uint32_t now) const { return have_ && elapsed(now, since_) > window_; }
 private:
  float delta_; uint32_t window_; float anchor_; uint32_t since_; bool have_;
};

// ---- 4. a stall detector: the motors are driven but the wheels do not turn ----
class StallDetector {
 public:
  StallDetector(float min_cmd, uint32_t window_ms) : min_cmd_(min_cmd), window_(window_ms
  void update(float cmd_speed, int32_t ticks, uint32_t now) {
    float c = cmd_speed < 0.0f ? -cmd_speed : cmd_speed;
    if (c < min_cmd_ || !have_ || ticks != last_ticks_) { last_ticks_ = ticks; since_ = n
  }
  bool stalled(uint32_t now) const { return have_ && elapsed(now, since_) > window_; }
 private:
  float min_cmd_; uint32_t window_; int32_t last_ticks_; uint32_t since_; bool have_;
};

// ---- 5. the supervisor: between the controller and the motors ----
struct SafetyConfig {
  uint32_t cmd_timeout_ms;      // a command older than this is dropped
  uint32_t range_stale_ms;      // no valid range reading for this long while moving forw
  float stuck_delta_m;          // a moving robot's reading must change by more than this
  uint32_t stuck_window_ms;     // ... within this window
  uint32_t stall_ms;            // driven but not turning for this long
  float stop_m;                 // never drive forward when the last valid range is below
  float v_max;                  // hard speed limit, m/s
};

class Supervisor {
 public:
  explicit Supervisor(const SafetyConfig& c)
      : c_(c), cmd_(c.cmd_timeout_ms), estop_(3), stuck_(c.stuck_delta_m, c.stuck_window_
        faults_(0), last_valid_(0), have_valid_(false), last_range_(-1.0f), moving_(false
  void command(const Cmd& c, uint32_t now) { cmd_.set(c, now); if (!cmd_seen_) { cmd_seen
  void estop_input(bool pressed) { estop_.sample(pressed); }
  bool reset() { if (!estop_.tripped() || estop_.reset()) { faults_ = 0; return true; } r
  // a new range measurement arrived (-1 = no echo): call this once per measurement, what
  void range_sample(uint32_t now, float range_m) {
```

```
    bool valid = range_m >= 0.0f;
    if (valid) { last_valid_ = now; have_valid_ = true; last_range_ = range_m; }
    stuck_.update(range_m, valid, moving_, now);
  }
  // one supervision step (call it on a fixed period, faster than the controller): retur
  Cmd step(uint32_t now, int32_t ticks_l, int32_t ticks_r) {
    Cmd req = cmd_.get(now);                                    // an expired command
    float fwd = (req.vl + req.vr) * 0.5f;
    moving_ = fwd > 0.05f || fwd < -0.05f;
    if (estop_.tripped()) faults_ |= F_ESTOP;
    if (fwd > 0.05f && elapsed(now, have_valid_ ? last_valid_ : first_cmd_) > c_.range_s
    if (stuck_.stuck(now)) faults_ |= F_STUCK_RANGE;
    stall_.update(fwd, ticks_l + ticks_r, now);
    if (stall_.stalled(now)) faults_ |= F_STALL;
    if (fwd > 0.05f && have_valid_ && last_range_ < c_.stop_m) faults_ |= F_TOO_CLOSE;
    Cmd out = req;
    if (faults_ != 0) { out.vl = 0.0f; out.vr = 0.0f; }
    if (out.vl > c_.v_max) out.vl = c_.v_max; else if (out.vl < -c_.v_max) out.vl = -c_.
    if (out.vr > c_.v_max) out.vr = c_.v_max; else if (out.vr < -c_.v_max) out.vr = -c_.
    return out;
  }
  uint8_t faults() const { return faults_; }
  bool cmd_expired(uint32_t now) const { return cmd_.expired(now); }
 private:
  SafetyConfig c_;
  ExpiringCmd cmd_;
  Estop estop_;
  StuckDetector stuck_;
  StallDetector stall_;
  uint8_t faults_;
  uint32_t last_valid_;
  bool have_valid_;
  float last_range_;
  bool moving_;
  bool cmd_seen_;
  uint32_t first_cmd_;
};
```

- **ExpiringCmd.** A command is only good for `timeout_ms` after it was given. If the controller hangs, crashes, or is simply late, the motors get zero, not the last command. (Expiry is not a latched fault: a fresh command restarts the robot.)

- **Estop.** A debounced, latched input. The switch must read "pressed" for `n` samples in a row to trip; releasing the switch does not restart the robot; `reset()` is refused while the switch is held.
- **StuckDetector.** While the robot is moving, a distance reading that does not change by more than `delta` within `window_ms` is not a measurement. It is not enough for it to be constant: a still robot sees a constant range.
- **StallDetector.** The motors are driven but the wheel ticks do not change.
- **Supervisor.** Sits between the controller and the motors. The controller only *asks* (`command`); the supervisor decides what reaches the motors (`step`): zero if any fault is latched, a stale range while driving forward, a wall closer than a stop distance, and a hard speed limit that applies whatever the controller says.

4. Fault injection on the simulator

A safety rule that has never been triggered is a hope. **Fault injection** breaks one thing at a chosen moment on the simulator and watches. Here a controller drives at a wall and stops 0.30 m from it, using the front range sensor; when it gets "no echo" it keeps its last command (a naive policy, on purpose, because that is what first versions do). At 2 s one thing breaks; each case is run without and with the supervisor, 20 runs each:

```
// e_faults.cpp -- fault injection on the simulator: a robot drives at a wall and one th
// The controller approaches the wall at up to 0.30 m/s and stops 0.30 m from it, using
// Each scenario is run without and with the supervisor of safety.h (command timeout 250
// Scored: runs that hit the wall (the front sensor reached the wall), mean time from th
#include <cstdio>
#include <cmath>
#include <initializer_list>
#include "sim.h"
#include "safety.h"

enum Scenario { NONE, HANG, STUCK_FAR, NO_ECHO, ENCODER_DEAD, WHEELS_BLOCKED, SENSOR_NOI
static const char* NAMES[] = {"no fault (control)", "controller hangs (stops sending com
                              "encoders dead (ticks frozen)", "wheels blocked (motors tu

struct R { bool hit; double stop_delay; double speed_at_end; bool stopped_by_sup; };

static R run(Scenario sc, bool supervised, uint32_t seed) {
  RobotParams rp; SensorParams sp; Arena arena;
```

```
  SimHW sim(rp, sp, arena, seed * 2654435761u + 12345u);          // a well-mixed seed: 
  SafetyConfig cfg = {250, 300, 0.02f, 500, 400, 0.15f, 0.4f};
  Supervisor sup(cfg);
  const double T_FAULT = 2.0, DT = 0.005;
  Cmd cmd = {0.0f, 0.0f}, applied = {0.0f, 0.0f};
  float last_range = 3.0f;
  double stop_time = -1; bool hit = false; bool sup_stopped = false;
  int32_t frozen_l = 0, frozen_r = 0; bool frozen = false;
  uint32_t last_ctrl = 0;
  for (long k = 0; k < 2000; ++k) {                                    // 10 s
    double t = k * DT; uint32_t now = (uint32_t)std::lround(t * 1000.0);
    bool fault = t >= T_FAULT;
    if (k % 12 == 0) {                                                 // the controlle
      bool hung = fault && sc == HANG;
      if (!hung) {
        float r = sim.range_m();
        if (fault && sc == STUCK_FAR) r = 2.5f;
        if (fault && sc == NO_ECHO) r = -1.0f;
        if (fault && sc == SENSOR_NOISY_BURST) r = 0.01f;
        if (supervised) sup.range_sample(now, r);
        if (r >= 0.0f) last_range = r;
        float v = last_range - 0.30f; if (v < 0.0f) v = 0.0f; if (v > 0.30f) v = 0.30f;
        cmd.vl = cmd.vr = v;
        if (supervised) sup.command(cmd, now);
        last_ctrl = now;
      } else if (supervised) {
        sup.range_sample(now, last_range);                             // the sensor st
      }
    }
    int32_t tl, tr; sim.read_ticks(tl, tr);
    if (fault && sc == ENCODER_DEAD) { if (!frozen) { frozen = true; frozen_l = tl; froz
    if (k % 4 == 0) {                                                  // the supervisor
      if (supervised) { applied = sup.step(now, tl, tr); if (sup.faults() && !sup_stoppe
      else applied = cmd;                                              // without it th
    }
    if (fault && sc == WHEELS_BLOCKED) { sim.drive(0.0f, 0.0f); }
    else sim.drive(applied.vl, applied.vr);
    sim.step(DT);
    if (sim.ray_distance() <= 0.0) hit = true;
    double speed = 0.5 * (sim.wheel_l() + sim.wheel_r());
    if (fault && stop_time < 0 && std::fabs(applied.vl) < 1e-6 && std::fabs(applied.vr) 
    (void)last_ctrl;
    if (hit) { R r = {true, stop_time < 0 ? -1.0 : stop_time - T_FAULT, speed, sup_stopp
  }
  R r = {false, stop_time < 0 ? -1.0 : stop_time - T_FAULT, 0.5 * (sim.wheel_l() + sim.wh
```

```
  return r;
}

int main() {
  std::printf("fault injection: a robot drives at a wall, one thing breaks at t = 2 s; 2(
  std::printf("  %-52s %-12s %10s   %24s   %s\n", "scenario", "supervisor", "hit wall",
  for (int s = 0; s < 7; ++s) {
    for (int sup = 0; sup < 2; ++sup) {
      int hits = 0, acted = 0, nstop = 0; double sum_delay = 0;
      for (uint32_t seed = 1; seed <= 20; ++seed) {
        R r = run((Scenario)s, sup != 0, seed);
        if (r.hit) ++hits;
        if (r.stopped_by_sup) ++acted;
        if (r.stop_delay >= 0) { sum_delay += r.stop_delay; ++nstop; }
      }
      char buf[40];
      if (s == NONE) std::snprintf(buf, sizeof buf, "-");
      else if (nstop) std::snprintf(buf, sizeof buf, "%.3f (%d of 20 stopped)", sum_delay
      std::printf("  %-52s %-12s %7d/20   %24s   %d/20\n", NAMES[s], sup ? "yes" : "no",
    }
  }
  return 0;
}
```

```
fault injection: a robot drives at a wall, one thing breaks at t = 2 s; 20 runs per line
  scenario                                          supervisor     hit wall   motors s
  no fault (control)                                no                 0/20
  no fault (control)                                yes                0/20
  controller hangs (stops sending commands)         no                20/20
  controller hangs (stops sending commands)         yes                0/20   0.240 (1
  range sensor stuck at 2.5 m                       no                20/20
  range sensor stuck at 2.5 m                       yes                0/20   0.560 (1
  range sensor sees no echo                         no                20/20
  range sensor sees no echo                         yes                0/20   0.297 (1
  encoders dead (ticks frozen)                      no                 0/20    7.780
  encoders dead (ticks frozen)                      yes                0/20   0.420 (1
  wheels blocked (motors turn, robot does not move) no                 0/20
  wheels blocked (motors turn, robot does not move) yes                0/20   0.699 (1
  range sensor stuck at 0.01 m (reads 'touching')   no                 0/20   0.040 (1
  range sensor stuck at 0.01 m (reads 'touching')   yes                0/20   0.040 (1
```

Without the supervisor, three faults drive the robot into the wall in every run: a hung controller (the last command stays on the motors), a range sensor stuck at a far value, and a range sensor that sees no echo (the naive policy keeps going). **With it, none does**: the motors are stopped 0.24 s after the hang (the 250 ms command timeout), 0.56 s after the sensor sticks (the 500 ms stuck window of this configuration plus the sampling), and 0.30 s after the echo is lost (the 300 ms stale limit). The stuck-at-0.01 m sensor is the instructive harmless one: the controller itself stops on a reading of "touching", so the fault is safe with or without the supervisor. The blocked wheels and the dead encoders hurt nothing with the walls here, and the supervisor stops them anyway (0.70 s and 0.42 s): a driven motor that does not turn is heating, and a robot that has lost its odometry is blind.

It also has a cost, and the table shows it. In the healthy control runs the supervisor stopped the robot in 2 of 20: the stuck detector fired during the slow creep up to the stopping distance, where the commanded speed is small and the range hardly changes. That is a *false alarm*, and a latched one: the robot stops short of its goal until someone resets it. Challenge 2 measures the trade between false alarms and detection delay.

Two mistakes the testing found in my own code

A start-up false latch. The first version of the stale-range rule latched a fault at time zero in 4 of 20 healthy runs, with the reading 3.000 m and the robot not yet moving: the rule asked for a valid reading, and the first measurement of those runs was a dropout. A rule that fires before the system has started is wrong; the fix is that, before the first valid reading, the clock for "stale" runs from the first command. **A bad seed.** Why 4 of 20, when dropouts are 2 %? The seeds were small multiples (77, 154, …) given to a xorshift generator, whose first outputs for small seeds are small numbers, which the simulator reads as a dropout. The runs above use well-mixed seeds. Both were found only because a healthy run was injected with *no* fault and the supervisor's reaction was counted.

5. The watchdog, brown-out and the stop button

The software supervisor cannot help when the software itself stops. That is the job of the **hardware watchdog** (Programming for Robotics with Arduino C++ 8): a timer that resets the chip unless the program feeds it, fed here only when every task has checked in since the last feed, so a wedged control task or echo task cannot be hidden by a loop that still runs. `robot_ch12.ino` turns the motors off *first* in `setup()`, before anything else can go wrong, and enables the watchdog at 250 ms.

On a board a supply that sags can corrupt the chip before the watchdog can do anything; the chip has a **brown-out detector** that holds it in reset below a voltage set by a fuse. The Uno's core says what the board is programmed with: in `boards.txt`, `uno.bootloader.extended_fuses=0xFD` and `uno.bootloader.high_fuses=0xDE`. That I verified. What those bytes mean, a brown-out level of about 2.7 V and the watchdog not being forced on by a fuse, is what I remember of the ATmega328P data sheet and *I did not check it*; read the data sheet before you rely on it, and note that a motor that stalls and sags the supply is exactly the case it is for.

An emergency stop that is code is a contradiction: code can only stop what the code can still control. A real stop must cut the motor power in **hardware**, with a switch in the supply, so that no software state can keep the motors running. In `robot_ch12.ino` the bumper switch on pin 11 feeds the latched `Estop` of `safety.h` as a stand-in, and the comment in the sketch says so.

6. A log that survives the fault

After a fault there is no room to print the whole history, and by then it is too late to ask what the robot was doing. A **ring buffer** keeps the last `N` events in a fixed array, overwriting the oldest, and is dumped when a fault latches. `RingLog` in `ringlog.h` stores a time, a code and a value (7 bytes on the AVR) and has no heap:

```
// ringlog.h -- an event log in a fixed ring buffer (Programming for Robotics with Arduin
// A robot that has just misbehaved cannot print everything it did: there is no room and
#pragma once
#include <stdint.h>

struct LogEvent { uint32_t t_ms; uint8_t code; int16_t value; };      // 7 bytes of data

template <uint8_t N>
class RingLog {
 public:
  RingLog() : head_(0), count_(0), total_(0) {}
  void add(uint32_t t_ms, uint8_t code, int16_t value) {
    buf_[head_].t_ms = t_ms; buf_[head_].code = code; buf_[head_].value = value;
    head_ = (uint8_t)((head_ + 1) % N);
    if (count_ < N) ++count_;
    ++total_;
  }
  uint8_t size() const { return count_; }
```

```
  uint32_t total() const { return total_; }                          // events ever add
  const LogEvent& at(uint8_t i) const { return buf_[(uint8_t)((head_ + N - count_ + i) %
 private:
  LogEvent buf_[N];
  uint8_t head_, count_;
  uint32_t total_;
};
```

How large `N` must be is a question with an answer you can compute, and challenge 3 does. First what all of it costs on the Uno. The cost-test sketch has seven variants (baseline, then one safety piece each, fed from `analogRead` and printed as an integer so the compiler cannot remove it):

```
// s_safe.ino -- cost test for chapter 12: the safety pieces on the Uno. Compile-checked
// VARIANT 0: baseline, 1: ExpiringCmd, 2: Estop, 3: StuckDetector, 4: StallDetector, 5:
#include "safety.h"
#include "ringlog.h"

#ifndef VARIANT
#define VARIANT 1
#endif

#if VARIANT == 1
static ExpiringCmd obj(250);
__attribute__((noinline)) int32_t step(float v, uint32_t now) { Cmd c = {v, v}; obj.set(
#elif VARIANT == 2
static Estop obj(3);
__attribute__((noinline)) int32_t step(bool pressed) { obj.sample(pressed); return obj.t
#elif VARIANT == 3
static StuckDetector obj(0.02f, 500);
__attribute__((noinline)) int32_t step(float r, uint32_t now) { obj.update(r, true, true
#elif VARIANT == 4
static StallDetector obj(0.05f, 400);
__attribute__((noinline)) int32_t step(float v, int32_t ticks, uint32_t now) { obj.updat
#elif VARIANT == 5
static const SafetyConfig SAFE = {250, 300, 0.02f, 1000, 400, 0.15f, 0.4f};
static Supervisor obj(SAFE);
__attribute__((noinline)) int32_t step(float v, float r, int32_t ticks, uint32_t now) {
#elif VARIANT == 6
static RingLog<24> obj;
__attribute__((noinline)) int32_t step(uint32_t now, int16_t v) { obj.add(now, 1, v); re
#endif
```

```
void setup() { Serial.begin(115200); }

void loop() {
  int a = analogRead(A0), b = analogRead(A1);      // inputs the compiler cannot know
  uint32_t now = millis();
  (void)a; (void)b; (void)now;                     // not every variant uses every input
#if VARIANT == 0
  Serial.println((int32_t)(a + b) * 1000L);
#elif VARIANT == 1
  Serial.println(step((float)a * 0.001f, now));
#elif VARIANT == 2
  Serial.println(step(a > 512));
#elif VARIANT == 3
  Serial.println(step((float)a * 0.003f, now));
#elif VARIANT == 4
  Serial.println(step((float)a * 0.001f, (int32_t)b, now));
#elif VARIANT == 5
  Serial.println(step((float)a * 0.001f, (float)b * 0.003f, (int32_t)b, now));
#elif VARIANT == 6
  Serial.println(step(now, (int16_t)a));
#endif
  delay(20);
}
```

```
variant                          flash (bytes)   vs baseline   RAM (bytes)   vs base
0  baseline (nothing)                 1982             -           188
1  ExpiringCmd                        2792          +810           205          +1
2  Estop                              2022           +40           192           +
3  StuckDetector                      3106         +1124           205          +1
4  StallDetector                      2890          +908           205          +1
5  Supervisor (all of it)             4432         +2450           315         +12
6  RingLog<24>                        2150          +168           362         +17
```

And, from `avr-objdump` (script `count_safe.py`), the function of each variant: size, instructions, and the library routines it calls:

```
ExpiringCmd: set + get              180 bytes   71 instructions   calls: __fixsfsi x1,
Estop: sample + tripped              64 bytes   23 instructions   calls: none
StuckDetector: update + stuck       294 bytes  109 instructions   calls: __cmpsf2 x2,
StallDetector: update + stalled     270 bytes  106 instructions   calls: __cmpsf2 x1,
```

```
Supervisor: command + sample + step  1198 bytes  449 instructions   calls: __addsf3 x1,
RingLog<24>: add                      116 bytes   44 instructions   calls: __divmodhi4 x
```

Most of the flash is **the floating-point library**, not the safety logic: `Estop` has no floating-point arithmetic, is 64 bytes and 23 instructions and costs +40 bytes of flash; the others pull in compare, multiply and subtract routines that the first of them pays for (+810 for `ExpiringCmd`, +1124 for `StuckDetector`, which are mostly the same routines counted again because each variant is built alone). In a sketch that already uses floats (every sketch of this course after the first chapters does) the marginal cost is the function itself: 180, 294, 270 and 1,198 bytes. The log costs RAM, not flash: 7 bytes an event, 168 for 24, plus 6 for its bookkeeping.

7. The robot

`robot_ch12.ino` is the robot of Programming for Robotics with Arduino C++ 8 (the Timer1 1 kHz tick, the non-blocking echo driver, the encoders on interrupts, the table-driven avoider) with the safety layer between the avoider and the motors. The avoider only *asks* for a command; the supervisor, stepped every 20 ms with the encoder ticks, decides what reaches the motors; every range measurement, every change of the applied command and every change of the fault mask goes into a 24-event ring; when a fault latches the ring is dumped over the serial line once; sending `r` over the serial line asks for a reset, which is refused while the stop button is held.

```
// robot_ch12.ino -- Programming for Robotics with Arduino C++, chapter 12: chapter 8's r
// a hardware watchdog, an emergency-stop input and a ring-buffer event log that is dump
// Compile-checked for the Uno (arduino:avr:uno). NOT run on a board: nothing was attach
// The bumper switch on pin 11 stands in for an emergency stop. A real emergency stop mu
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <util/atomic.h>
#include "conv.h"
#include "echo.h"
#include "pins.h"
#include "filters.h"
#include "motion.h"
#include "fsm.h"
#include "wdt.h"
#include "safety.h"
#include "ringlog.h"
```

```
const float V_MAX_MPS = 0.5f;
const uint8_t DEAD_DUTY = 40;                          // ASSUMED: measure yours
const float PER_TICK = 6.2831853f * 0.033f / 360.0f;

volatile uint32_t g_ms = 0;
volatile uint8_t g_control_due = 0, g_echo_due = 0, g_report_due = 0;
volatile int32_t g_ticks_l = 0, g_ticks_r = 0;

ISR(TIMER1_COMPA_vect) {                               // every 1 ms: only counts and se
  static uint8_t c20 = 0, c60 = 0;
  static uint16_t c500 = 0;
  ++g_ms;
  if (++c20 >= 20) { c20 = 0; g_control_due = 1; }
  if (++c60 >= 60) { c60 = 0; g_echo_due = 1; }
  if (++c500 >= 500) { c500 = 0; g_report_due = 1; }
}
ISR(INT0_vect) { bool a = (PIND & _BV(PD2)) != 0, b = (PINC & _BV(PC1)) != 0; g_ticks_l
ISR(INT1_vect) { bool a = (PIND & _BV(PD3)) != 0, b = (PINC & _BV(PC2)) != 0; g_ticks_r

static uint32_t now_ms() { uint32_t t; ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { t = g_ms; } r
static void timer1_begin_1khz() { TCCR1A = 0; TCCR1B = _BV(WGM12) | _BV(CS10); OCR1A = 1

static void drive(float vl, float vr) {
  digitalWrite(PIN_ML_DIR, vl >= 0.0f ? HIGH : LOW);
  digitalWrite(PIN_MR_DIR, vr >= 0.0f ? HIGH : LOW);
  analogWrite(PIN_ML_PWM, speed_to_pwm_dz(vl >= 0.0f ? vl : -vl, V_MAX_MPS, DEAD_DUTY));
  analogWrite(PIN_MR_PWM, speed_to_pwm_dz(vr >= 0.0f ? vr : -vr, V_MAX_MPS, DEAD_DUTY));
}

// the controller: as in chapter 8
static AvoidConfig CFG = {0.3f, 0.25f, 0.2f, 2.0f, 0.15f, 400, 400, 1200, 3, true};
static TableAvoider machine(CFG, 12345);
static Median<int16_t, 5> median;
static Echo echo;
static Heartbeats<2> beats;
static bool trig_level = false;
static float g_range = -1.0f;

// the safety layer
static const SafetyConfig SAFE = {250, 300, 0.02f, 1000, 400, 0.15f, 0.4f};     // from
static Supervisor sup(SAFE);
static RingLog<24> logbuf;                                                       // 24 ev
static uint8_t last_mask = 0;
static bool dumped = false;
enum : uint8_t { EV_RANGE = 1, EV_CMD = 2, EV_FAULT = 3, EV_ESTOP = 4 };
```

```
static float last_v = -9.0f;

static void dump_log() {
  Serial.print(F("-- log: ")); Serial.print(logbuf.size()); Serial.print(F(" of ")); Ser
  for (uint8_t i = 0; i < logbuf.size(); ++i) {
    const LogEvent& e = logbuf.at(i);
    Serial.print(e.t_ms); Serial.print(' '); Serial.print(e.code); Serial.print(' '); Se
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_ML_DIR, OUTPUT); pinMode(PIN_MR_DIR, OUTPUT); pinMode(PIN_ML_PWM, OUTPUT);
  pinMode(PIN_TRIG, OUTPUT); pinMode(PIN_ECHO, INPUT); pinMode(PIN_BUMP, INPUT_PULLUP);
  pinMode(PIN_ENC_L, INPUT_PULLUP); pinMode(PIN_ENC_R, INPUT_PULLUP); pinMode(PIN_ENC_L_
  drive(0.0f, 0.0f);                                  // the motors are off BEFORE anyt
  EICRA = _BV(ISC00) | _BV(ISC10); EIFR = _BV(INTF0) | _BV(INTF1); EIMSK = _BV(INT0) | _
  timer1_begin_1khz();
  wdt_enable(WDTO_250MS);
}

void loop() {
  bool before = trig_level;
  Echo::Status st = echo.update(micros(), digitalRead(PIN_ECHO), trig_level);
  if (trig_level != before) digitalWrite(PIN_TRIG, trig_level);
  bool new_range = false;
  if (st == Echo::READY) { g_range = (float)median.update((int16_t)(echo_us_to_m(echo.wi
  else if (st == Echo::TIMEOUT) { g_range = -1.0f; new_range = true; }
  if (new_range) { sup.range_sample(now_ms(), g_range); logbuf.add(now_ms(), EV_RANGE, (

  if (g_echo_due) { g_echo_due = 0; echo.start(); beats.checkin(1); }
  if (g_control_due) {
    g_control_due = 0;
    uint32_t now = now_ms();
    bool bump = digitalRead(PIN_BUMP) == LOW;
    sup.estop_input(bump);
    Sense s; s.front_m = g_range; s.right_m = -1.0f; s.bump = bump;
    sup.command(machine.update(s, now), now);          // the controller only ASKS; the
    int32_t l, r; ATOMIC_BLOCK(ATOMIC_RESTORESTATE) { l = g_ticks_l; r = g_ticks_r; }
    Cmd out = sup.step(now, l, r);
    drive(out.vl, out.vr);
    if (out.vl != last_v) { logbuf.add(now, EV_CMD, (int16_t)(out.vl * 1000.0f)); last_v
    if (sup.faults() != last_mask) { logbuf.add(now, EV_FAULT, (int16_t)sup.faults()); la
    beats.checkin(0);
  }
```

```
    if (sup.faults() != 0 && !dumped) { dump_log(); dumped = true; }                // af
    if (Serial.available() && Serial.read() == 'r') { if (sup.reset()) { last_mask = 0; du
    if (g_report_due) {
      g_report_due = 0;
      Serial.print(now_ms()); Serial.print(F(" ms  range ")); Serial.print(g_range, 2); Se
    }
    if (beats.all_alive()) { wdt_reset(); beats.clear(); }
  }
```

It compiles for the Uno under `--warnings all` with no warning from the sketch to:

```
Sketch uses 9692 bytes (30%) of program storage space. Maximum is 32256 bytes.
Global variables use 644 bytes (31%) of dynamic memory, leaving 1404 bytes for local var
```

That is 9,692 bytes of flash and 644 bytes of RAM: 2,774 bytes of flash and 317 bytes of RAM more than the robot of Programming for Robotics with Arduino C++ 8 (6,918 and 327), of which 168 bytes of RAM are the log. It was **not run**: nothing was attached, the watchdog was never tripped, the supervisor's thresholds are those of the simulator and the hardware may need others, and the false alarm of section 4 would show up on a real robot too.

8. Where to go next

This is from memory and nothing in it was run for this course; treat it as a list of questions to check against the data sheets, not as results. The Uno R3 was chosen because its limits make the lessons visible: 2 KB of RAM, 32 KB of flash, 16 MHz, no floating-point unit. Many later boards remove one limit at a time: more RAM and flash, a floating-point unit in the 32-bit ARM parts (which would change the float-against-fixed question of Programming for Robotics with Arduino C++ 10), several cores or radios in the ESP32 and the Pico. What does *not* go away is the method of the course: write the logic as plain C++ behind a small hardware interface, test it on the PC, and count what it costs.

Two steps up from the cooperative scheduler of Programming for Robotics with Arduino C++ 8: a **real-time operating system** (FreeRTOS is the common one) gives preemptive tasks with priorities, so a slow task can no longer delay a fast one, at the price of a stack for each task and the need for locks around shared data; and **micro-ROS** puts a small ROS 2 node on a microcontroller, so the board can talk to the rest of a robot as one more sensor or actuator. The

supervisor of this chapter does not become unnecessary on any of them: the safe state is still stopped and the guard still must not depend on the thing it guards.

9. The course in one table

The running theme of the course was the budget. Here is what each chapter's whole-robot sketch compiled to for the Uno (Programming for Robotics with Arduino C++ 1 has only size tests, and no whole-robot sketch). All are compile-time figures from the compiler; none was run on a board.

```
  chapter  sketch            flash (bytes)    RAM (bytes)    what it added
    2      robot_ch2.ino           9,606            936       talking to the hardware
    3      robot_ch3.ino           7,726            413       a simulator to test in
    4      robot_ch4.ino           8,626            433       sensor code and filtering
    5      robot_ch5.ino           8,868            409       driving and odometry
    6      robot_ch6.ino           7,638            541       control loops
    7      robot_ch7.ino           7,114            381       state machines
    8      robot_ch8.ino           6,918            327       interrupts, timing, the watchd
    9      robot_ch9.ino          11,388            943       planning in kilobytes
   10      robot_ch10.ino          9,726            574       state estimation
   11      robot_ch11.ino         10,040            437       robot arms
   12      robot_ch12.ino          9,692            644       the safety supervisor and the
  (Uno R3: 32,256 bytes of flash after the bootloader, 2,048 bytes of RAM.)
```

The robot never needed more than 11,388 bytes of flash (35 %) or 943 bytes of RAM (46 %), so the Uno was *enough*; where it was tight was RAM (the chapter 9 planner left 61 bytes of stack) and, for anything with floats, the library. And an honest summary of the status of the course: **the logic was run** on a PC, against simulated sensors whose numbers were my assumptions; **the sketches were compiled** for the Uno and their sizes are the compiler's; **no sketch was ever run on a board**, no time was ever measured, no sensor, motor, gyro or servo was ever connected. The first thing to do with a real robot is to take each assumption of this course and replace it with a measurement.

10. What the numbers can and cannot be trusted for

What they show. The tests pass, and a suite that passes can be shown to fail: 20 of 21 mutants killed after the first run showed 14. Three faults that crash a naive robot into a wall in every run

do not once the supervisor is between the controller and the motors. Expiring outputs, a latched stop, stuck and stall detectors and a log are small: 64 to 1,198 bytes of code each. The costs on the Uno are bytes, instructions and library calls.

What they do not show. The faults are the ones I thought of; a robot fails in ways nobody thought of. The thresholds were tuned on a simulator and may be wrong on a robot. The supervisor itself can have bugs, and it is checked only by these tests. The tests exercise the host build; the interrupts, the pins, the watchdog and the brown-out detector were never tested. **No time was measured**: "the motors are stopped 0.24 s after the hang" is simulated time with an assumed motor lag, not a stopping time on a robot. `robot_ch12.ino` was compiled, not run.

CHALLENGE 1: HOW LONG MAY A COMMAND LIVE?

Sweep the command timeout of the supervisor from 50 ms to 4 s. For each value find how far the robot travels after the controller hangs while driving at cruise speed toward a wall 3 m from the start (the hang happens at 6.5 s into the run), and, in a run with no fault but a controller that is sometimes late (5 % of its runs 150 ms late, 1 % 400 ms late), how often the timeout expires a command that was only late. Choose a timeout and say what limits it from above and from below.

View solution

CHALLENGE 2: THE STUCK DETECTOR'S THRESHOLDS

For a change threshold of 10, 20 and 40 mm and a window of 250 ms to 4 s, run the supervisor on 100 healthy approaches to the wall (they end with a slow creep) and on 100 runs where the range sensor sticks at 2.5 m. Report the false alarms and the detection delay for every pair. Which pair would you use, and what does the delay cost in distance travelled at 0.3 m/s?

View solution

CHALLENGE 3: HOW BIG MUST THE EVENT LOG BE?

Log every range measurement, every change of the applied command and every change of the fault mask while the range sensor sticks at 2 s and the fault latches. For ring buffers of 8 to 255 events report how much history the dump holds, whether it reaches back past the moment the sensor stuck, and what it costs in RAM at 7 bytes an event. Choose a size.

View solution

QUICK REFERENCE

- **Unit tests.** A 20-line harness (`CHECK`, `CHECK_NEAR`); 23 tests, 1,360 checks on the logic of the course; expected numbers come from maths and definitions, not from the code's own output.
- **Mutation testing.** One small change per mutant; killed = a check fails. First run: 14 of 21 killed (7 holes in the tests); after the new tests 20 of 21; the survivor is an equivalent mutant. A green suite means nothing until it has failed on purpose.
- **Principles.** The safe state is stopped; the guard must not depend on what it guards; faults latch until a person resets.
- **Pieces.** ExpiringCmd (zero after the timeout), Estop (debounced, latched, reset only when released), StuckDetector (moving and no change), StallDetector (driven and no ticks), Supervisor (between controller and motors, speed limit), RingLog (last N events).
- **Fault injection (simulated, 20 runs).** Hung controller, stuck range, no echo: 20/20 hit the wall without the supervisor, 0/20 with it; motors stopped after 0.24, 0.56 and 0.30 s. It also stopped 2 of 20 healthy runs (a false alarm in the final creep).
- **Timeout (challenge 1).** 4 s: 50 of 50 hit the wall; 2 s: 0.62 m after a hang; 500 ms: 0.17 m and no spurious expiries with a controller up to 400 ms late; 250 ms: 1.3 spurious expiries per 10 s.
- **Stuck detector (challenge 2).** 20 mm and 500 ms: 10 false alarms in 100 healthy runs; 20 mm and 1 s: none, detected after 1.06 s (0.32 m at 0.3 m/s); 250 ms: 34 to 100 false alarms.
- **Log (challenge 3).** About 19 events a second near the fault; a log of 32 events (224 bytes) reaches back 0.68 s past the moment the sensor stuck, 64 (448 bytes) holds the whole 3 s run.
- **Hardware.** Watchdog fed only when every task has checked in; motors off first in `setup()`; a real emergency stop cuts power in hardware; Uno fuses 0xFD and 0xDE (read from `boards.txt`; their meaning from memory, unchecked).
- **Cost on the Uno (compiled, not run).** Supervisor +2,450 bytes flash alone (mostly the float library), +127 RAM; Estop +40 flash; RingLog<24> 174 RAM. `robot_ch12.ino`: 9,692 bytes flash (30 %), 644 bytes RAM (31 %).
- **Not run:** anything on a board; the watchdog, the brown-out detector or the stop button; any real fault; any time.

Course complete

That is the last chapter of Programming for Robotics with Arduino C++: twelve chapters, from the first `setup()` and `loop()` to a robot with a safety supervisor. The companion course, Programming for Robotics, covers the same topics in Python, in the same order. What comes

next is the part this course could not do: put a robot on the table, measure what it really does, and replace every number that this course marked as an assumption.