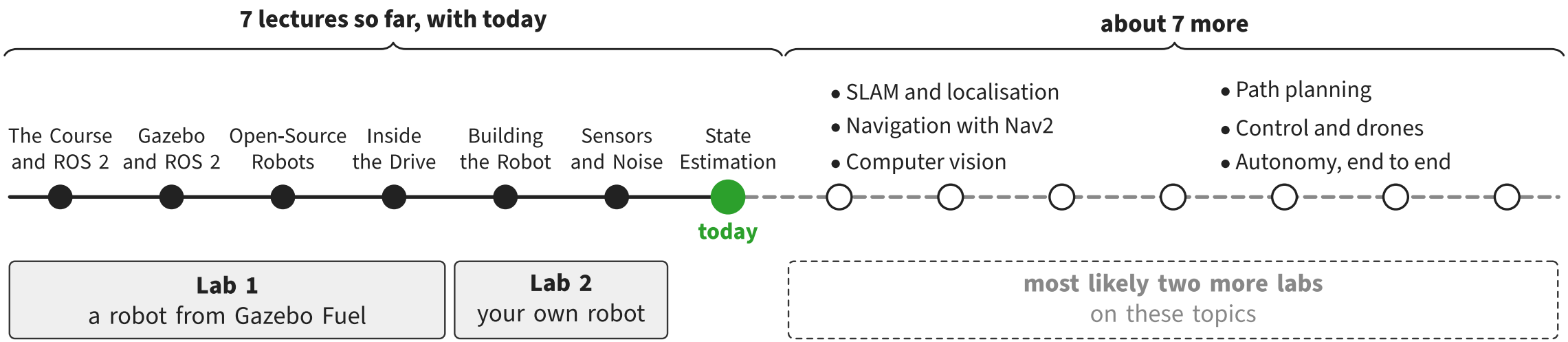
AUTONOMOUS INTELLIGENCE GROUP

LECTURE 6 — STATE ESTIMATION AND FUSION (I)

ONE ESTIMATE FROM NOISY SENSORS: FROM THE COMPLEMENTARY FILTER TO KALMAN BY HAND

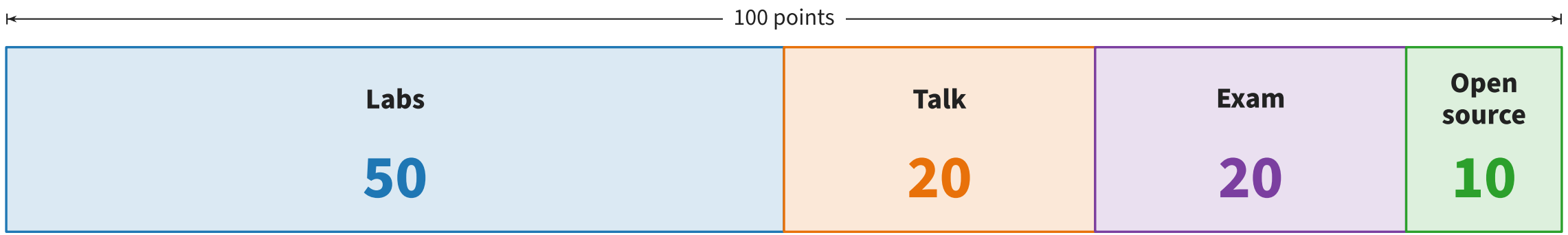
Topics: Course feedback · Lab 2 · Complementary filter · Bayes filter · Kalman filter

HALFWAY THROUGH THE COURSE



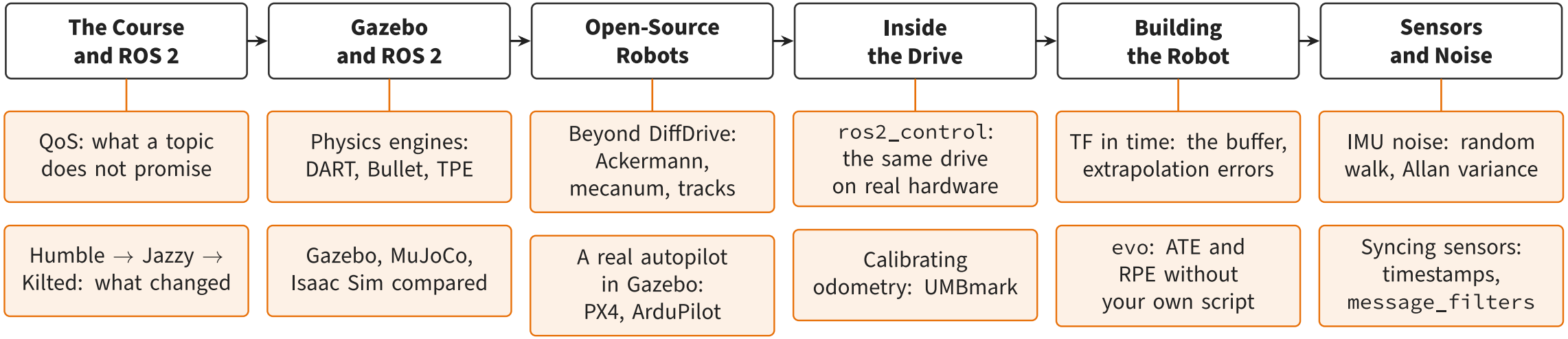
- **Lab 1** — a ready-made robot from Gazebo Fuel, driven from ROS 2: commands in, odometry out, its path plotted
- **Lab 2** — your own robot in URDF with a lidar and an IMU: noise from a real datasheet, checked against ground truth
- **Ahead** — about 7 more lectures on these topics, in rough order; most likely two more labs on them

GRADING: A REMINDER



- **Labs** — the README is what you hand in: how to run it, what you see, your numbers
- **Talk** — 15–20 min on a topic of your choice, agreed with me
- **Exam** — a 15–20 min interview: your results, your decisions and where they stop working
- **Open source** — a repository and its issues, reviewed: where you could contribute

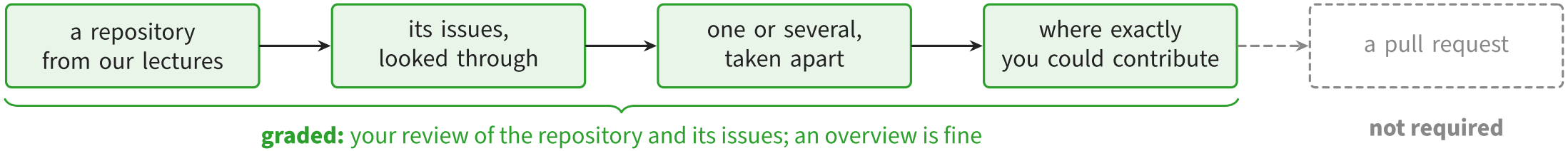
TALK IDEAS: TOPICS WE ONLY TOUCHED



15–20 min, your topic, agreed with me. What stayed unclear in your `notes.txt` can be a topic too.

📖 roscon.ros.org · 📖 docs.ros.org/en/rolling/Releases.html

OPEN SOURCE: REPOSITORIES WE USED



LECTURE	🎯 PUBLIC REPOSITORIES WE USED
The Course and ROS 2	github.com/ros2/common_interfaces · github.com/ROBOTIS-GIT/turtlebot3
Gazebo and ROS 2	github.com/gazebosim/gz-physics · github.com/ros2/rviz
Open-Source Robots	github.com/gazebosim/ros_gz · github.com/ros2/teleop_twist_keyboard
Inside the Drive	github.com/gazebosim/gz-sim · github.com/clearpathrobotics/clearpath_common
Building the Robot	github.com/ros/robot_state_publisher · github.com/MichaelGrupp/evo
Sensors and Noise	github.com/SteveMacenski/slam_toolbox · github.com/MOGI-ROS/Week-5-6-Gazebo-sensors
State Estimation	github.com/cra-ros-pkg/robot_localization · github.com/CCNYRoboticsLab/imu_tools

TODAY: THE COMPLEMENTARY FILTER, BAYES, KALMAN, THE EKF

1 · The Complementary Filter → **2 · Noise and the Average** → **3 · The Bayes Filter: Predict and Update** → **4 · The Kalman Filter in 1D**

Two poor sensors, one good angle: mostly the gyroscope, a little of the accelerometer

A guess and its doubt: averaging shrinks the doubt, but lags when the robot moves

How a scan changes what the robot believes: motion blurs it, a scan sharpens it

The robot drives to the wall: predict with the wheels, update with the lidar, the best weight at every step

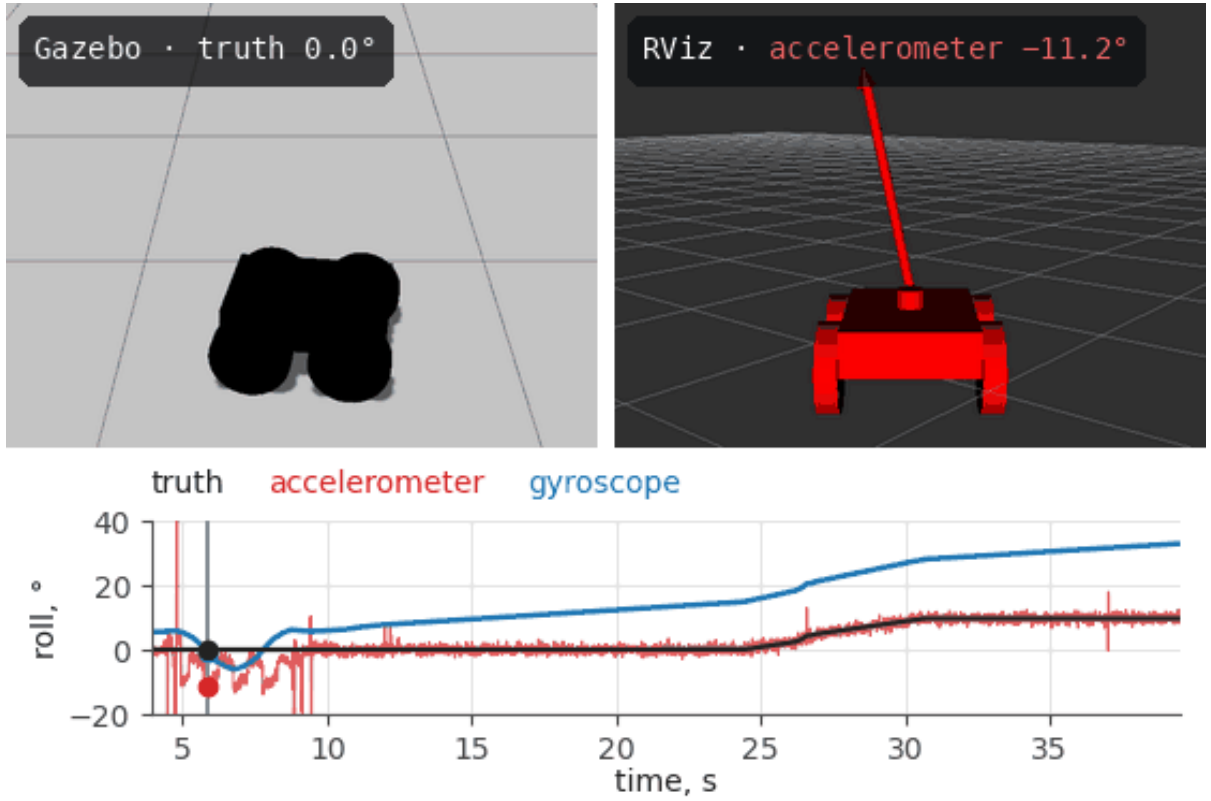
5 · The State and Its Covariance → **6 · When the Robot Turns: the EKF** → **7 · Fusing Wheels and Gyroscope**

A robot on the floor: position and heading, and errors that depend on each other

A straight-line model loses a turning robot; the extended Kalman filter follows the curve

Our robot in ROS 2: the wheels slip in turns, the gyroscope drifts, one filter fuses both

THE GYROSCOPE DRIFTS, THE ACCELEROMETER SHAKES



- **The gyroscope** is smooth, but its small error adds up: it drifts
- **The accelerometer** does not drift, but it shakes, and in a turn it shows a tilt that is not there
- **Together:** each is good where the other is poor — so we fuse them

LEGEND

tilt — how far the robot leans: the angle between its up axis and the vertical

roll — the robot's tilt to the side, °

black line — the true roll, from Gazebo

blue line — the roll from the gyroscope

red line — the roll from the accelerometer

grey vertical line — the moment the pictures show

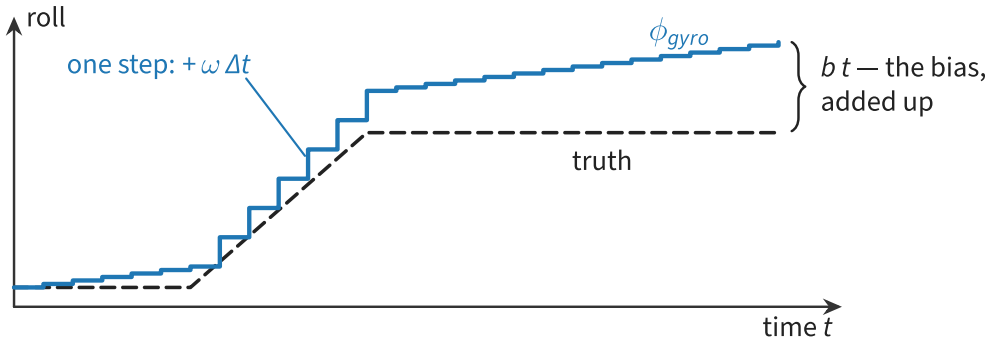
red arrow in RViz — what the accelerometer feels; RViz holds the robot model level

dark boxes — the roll at this moment: the truth in Gazebo, the accelerometer's in RViz

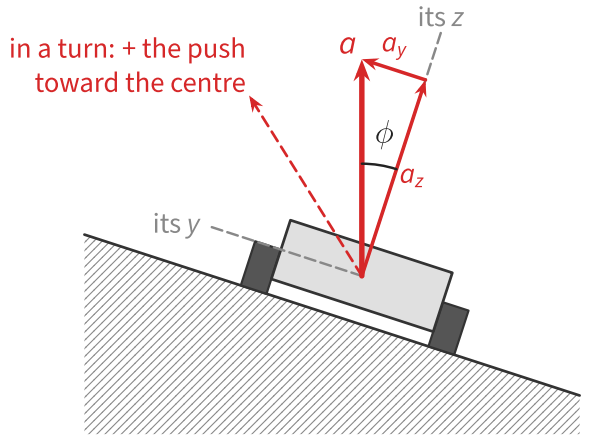
📖 porabook.com/book/

HOW EACH SENSOR MEASURES THE ROLL

gyroscope: add up its rate, step by step



accelerometer: the tilt of gravity's push



standing still, it feels only gravity's push *a*, straight up in the world

split along the robot's axes: the angle between *a* and its *z* is the roll ϕ_{acc}

s3.amazonaws.com/scolton-www/docs/filter.pdf

Each sensor's own roll — the ingredients of the filter:

Gyroscope — its rate, added up step by step:

$$\phi_{gyro} = \Sigma\, \omega\, \Delta t$$

smooth, but it **drifts**: its bias b adds up too, $b\,t$ after a time t

Accelerometer — the tilt of gravity's push:

$$\phi_{acc} = \mathrm{atan2}(a_y, a_z)$$

no drift, but it **shakes**, and in a turn it "tilts" sideways without tilting

LEGEND

ϕ_{gyro}, ϕ_{acc} — the roll from the gyroscope and from the accelerometer, rad

ω — the turn rate from the gyroscope, rad/s

Δt — one IMU step: 0.01 s at 100 Hz

b — the gyroscope's bias: the same small error in every reading

t — the time since the start, s

a_y, a_z — the accelerometer along the robot's own axes: sideways and up, m/s²

$\mathrm{atan2}$ — the angle of a vector from its two parts, with the right sign

blue steps — the gyroscope's sum; dashed — the truth

red arrows — gravity's push a and its two parts

THE GYROSCOPE DRIFTS, THE ACCELEROMETER SHAKES.
HOW DO WE GET ONE GOOD ANGLE FROM THE TWO?

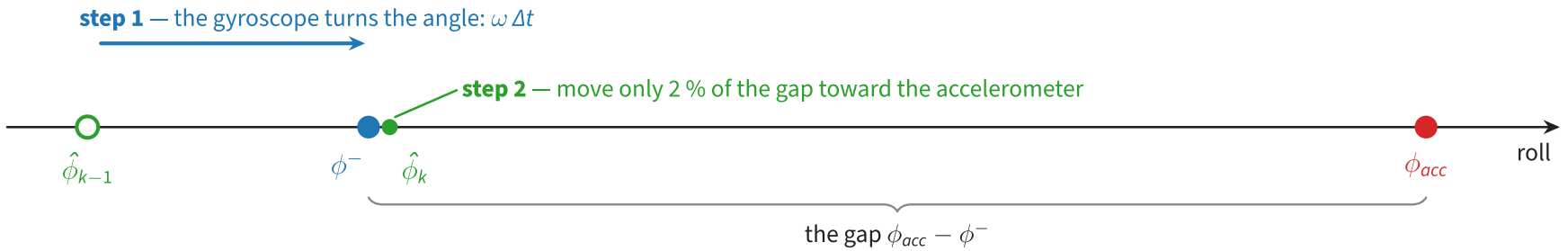
ANSWER: THE COMPLEMENTARY FILTER — 98 % GYROSCOPE, 2 % ACCELEROMETER

Step 1 — predict with the gyroscope: $\phi^- = \hat{\phi}_{k-1} + \omega\,\Delta t$

Step 2 — pull 2 % toward the accelerometer:

$$\hat{\phi}_k = \alpha\,\phi^- + (1-\alpha)\,\phi_{acc}, \qquad \alpha = 0.98$$

The gyroscope's drift is pulled back a little every step; the accelerometer's shaking barely gets in.



youtu.be/whSw42XddsU
ahrs.readthedocs.io/en/latest/filters/complementary.html

LEGEND

ϕ^- — the prediction: the old estimate turned by the gyroscope; the minus — before the measurement corrects it, the usual mark of a prediction

$\hat{\phi}_{k-1}$ — the old estimate; the hat — an estimate, not the truth;

k — the step number

$\omega\,\Delta t$ — the gyroscope's turn in one step: rate ω × one IMU step Δt, 0.01 s

ϕ_{acc} — the accelerometer's roll: the angle of gravity's push from its two parts, $\operatorname{atan2}(a_y, a_z)$

a_y, a_z — the accelerometer along the robot's own axes: sideways and up, m/s²

$\hat{\phi}_k$ — the new estimate

the gap $\phi_{acc} - \phi^-$ — how far the accelerometer is from the prediction

α — the weight of the prediction, between 0 and 1;

$1 - \alpha$ — the accelerometer's

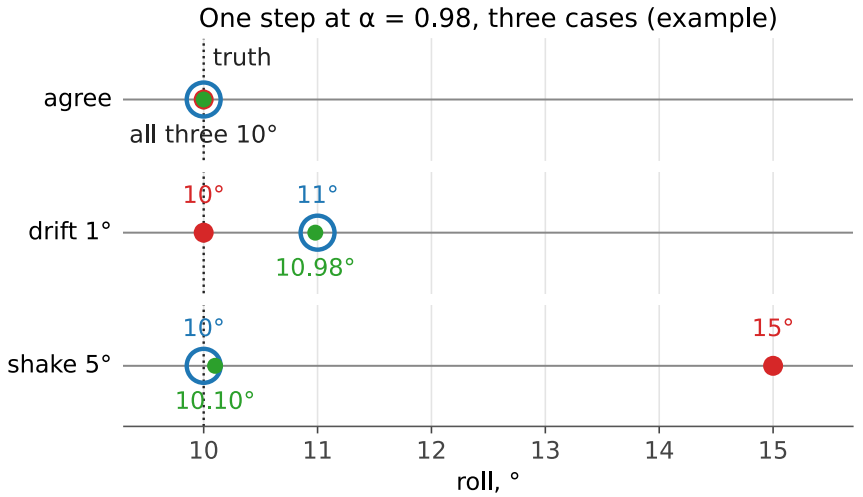
THE GAP IN THREE CASES: AGREE, DRIFT, SHAKE

The same formula, written with the gap between the two angles:

$$\hat{\phi}_k = \phi^- + 0.02\,(\phi_{acc} - \phi^-)$$

The robot stands still on the slope; the true roll is 10° (example):

CASE	ϕ^-	ϕ_{acc}	THE GAP	$\hat{\phi}_k$
agree	10°	10°	0	**10°**
drift 1°	11°	10°	−1°	**10.98°**
shake 5°	10°	15°	+5°	**10.1°**



LEGEND

blue ring — ϕ^-, the gyroscope's prediction: the old estimate plus one gyroscope step, not its whole sum

red dot — ϕ_{acc}, the accelerometer's roll: the angle of gravity's push from its two parts, $\mathrm{atan2}(a_y, a_z)$

a_y, a_z — the accelerometer along the robot's own axes: sideways and up, m/s²

green dot — $\hat{\phi}_k$, the new estimate; k — the step number

dotted line — the truth, 10° in all three cases

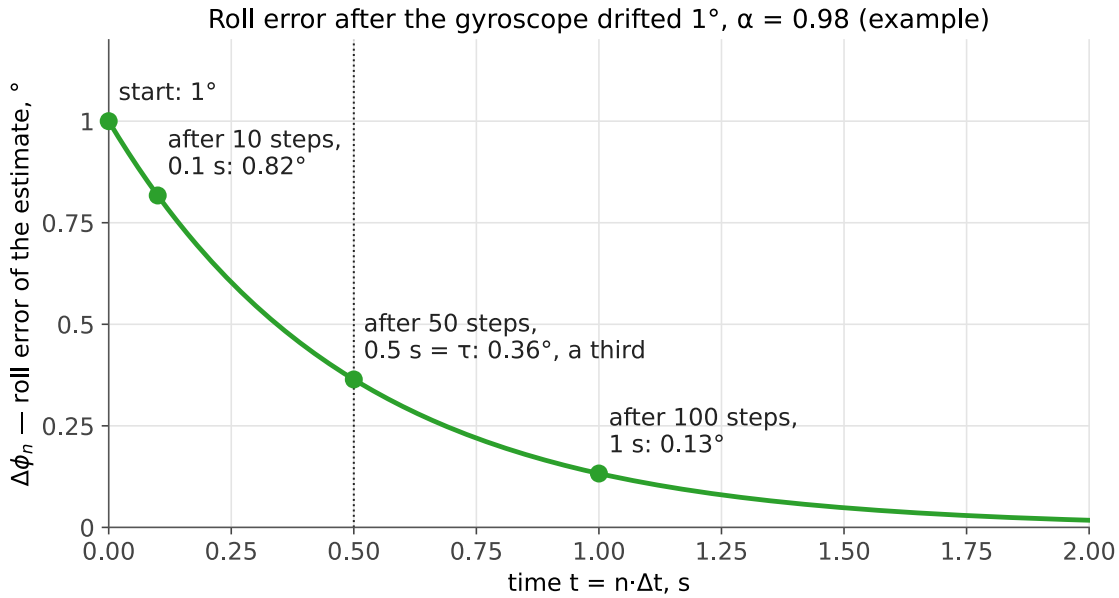
the gap $\phi_{acc} - \phi^-$ — how far the accelerometer is from the prediction

0.02 — the accelerometer's share, $1 - \alpha$, with $\alpha = 0.98$

- **Agree** — no gap, nothing changes: the weights add up to 1
- **Drift** — the gyroscope is 1° off; one step takes 2 % of it away: 0.02°
- **Shake** — the accelerometer jumps 5°; only 2 % of it gets in: 0.1°

s3.amazonaws.com/scolton-www/docs/filter.pdf

THE TIME CONSTANT: HOW LONG AN OLD ERROR LASTS



Each step, the filter:

$$\hat{\phi}_k = \alpha\,\phi^{-} + (1-\alpha)\,\phi_{acc}$$

An old error after n steps:

$$\Delta\phi_n = \alpha^n\,\Delta\phi_0$$

The time constant:

$$\tau \approx \frac{\Delta t}{1-\alpha} = \frac{0.01\ \text{s}}{0.02} = 0.5\ \text{s}$$

How long the filter remembers an old error: after τ a third of it is left.

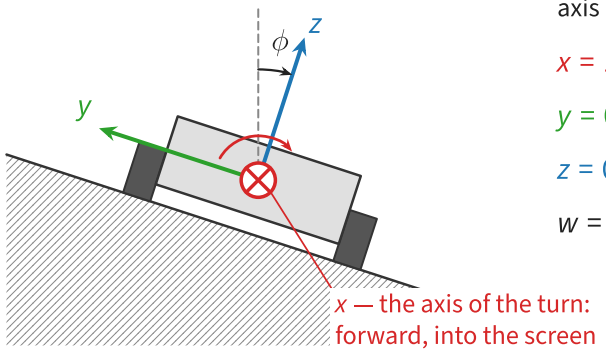
LEGEND

green curve — $\Delta\phi_n$ over time; dots — the start and after 10, 50, 100 steps
dotted line — the moment $t = \tau$
$\Delta\phi_n$ — the roll error after n steps: the estimate minus the true roll, °; $\Delta\phi_0$ — at the start, 1° here
$\hat{\phi}_k$ — the new estimate; k — the step number
ϕ^{-} — the gyroscope's prediction
ϕ_{acc} — the accelerometer's roll
α — the prediction's weight, 0.98 here; $1-\alpha$ — the accelerometer's share
n — the number of steps
τ — the time constant, s
Δt — one IMU step: 0.01 s at 100 Hz

- Only the prediction ϕ^{-} carries the error, weight α: 1° → 0.98° → 0.96° … (example)
- A smaller share of the accelerometer, a longer memory: $1-\alpha$ = 0.01 gives about 1 s
- Changes slower than τ come from the accelerometer, faster ones from the gyroscope

s3.amazonaws.com/scolton-www/docs/filter.pdf

A QUATERNION: ONE ROTATION IN FOUR NUMBERS



this turn as a quaternion, axis (1, 0, 0), angle ϕ:

$x = 1 \cdot \sin\frac{\phi}{2}$

$y = 0 \cdot \sin\frac{\phi}{2} = 0$

$z = 0 \cdot \sin\frac{\phi}{2} = 0$

$w = \cos\frac{\phi}{2}$

- **Any orientation is one turn** about one axis, set by the turn itself: the line it leaves in place — x for a roll, z for a yaw
- **A quaternion** keeps that axis and angle in four numbers, as in `Imu`, `/odom`, `tf2_echo`:

$$(x,\ y,\ z) = \text{axis} \cdot \sin\frac{\text{angle}}{2} \qquad w = \cos\frac{\text{angle}}{2}$$

One rotation in four numbers, and two turns combine by multiplying.

- **Why not roll, pitch, yaw:** order matters, they jump at ±180°, and at a pitch of 90° roll and yaw merge
- **On our slope the robot rolls about x**, forward: axis $(1,\ 0,\ 0)$, angle ϕ = 9.7°, so $y = z = 0$:
  x = sin 4.85° = 0.085, w = cos 4.85° = 0.996
- No rotation: $(0, 0, 0, 1)$

LEGEND

axis — the line the turn leaves in place, as three numbers along x, y, z; for the roll $(1, 0, 0)$ — x, forward

angle — how far it turns; here the roll ϕ = 9.7°, from the dashed vertical to the robot's z

x, y, z, w — the four numbers of a quaternion, as ROS names them; x, y, z go with the axes x, y, z

$\sin, \cos$ of half the angle — a quaternion keeps half of it: 4.85°

red ⊗, **green**, **blue** — the robot's axes x (into the screen), y, z, as RViz colours them, and their numbers in the quaternion

📖 docs.ros.org/en/jazzy/Tutorials/Intermediate/Tf2/Quaternion-Fundamentals.html

FORMULA ↔ IMU_TOOLS CODE

$$\hat{\phi}_k = \alpha\,\phi^{-} + (1-\alpha)\,\phi_{acc}$$

- $\hat{\phi}_k$ — the new estimate, k — the step · ϕ^{-} — the gyroscope's prediction · ϕ_{acc} — the accelerometer's roll · α — the prediction's weight
- `getPrediction` — step 1, turn by the gyroscope
- `getAccCorrection` — how far to the accelerometer's "down"; `scaleQuaternion` — keeps only the share `gain` of that turn; `gain` $= 1 - \alpha$ (default `0.01`, so $\alpha = 0.99$)
- `quaternionMultiplication` — the new estimate: the prediction turned by that share
- `w…`, `a…` — the gyroscope's and the accelerometer's readings, `dt` — one IMU step Δt; `q…` — the orientation as a quaternion (four numbers of one rotation): the mix is a turn, the same idea in 3D; `qI` — no turn, $(0, 0, 0, 1)$

C++ · imu_complementary_filter/src/complementary_filter.cpp — the filter inside imu_tools' complementary_filter_node

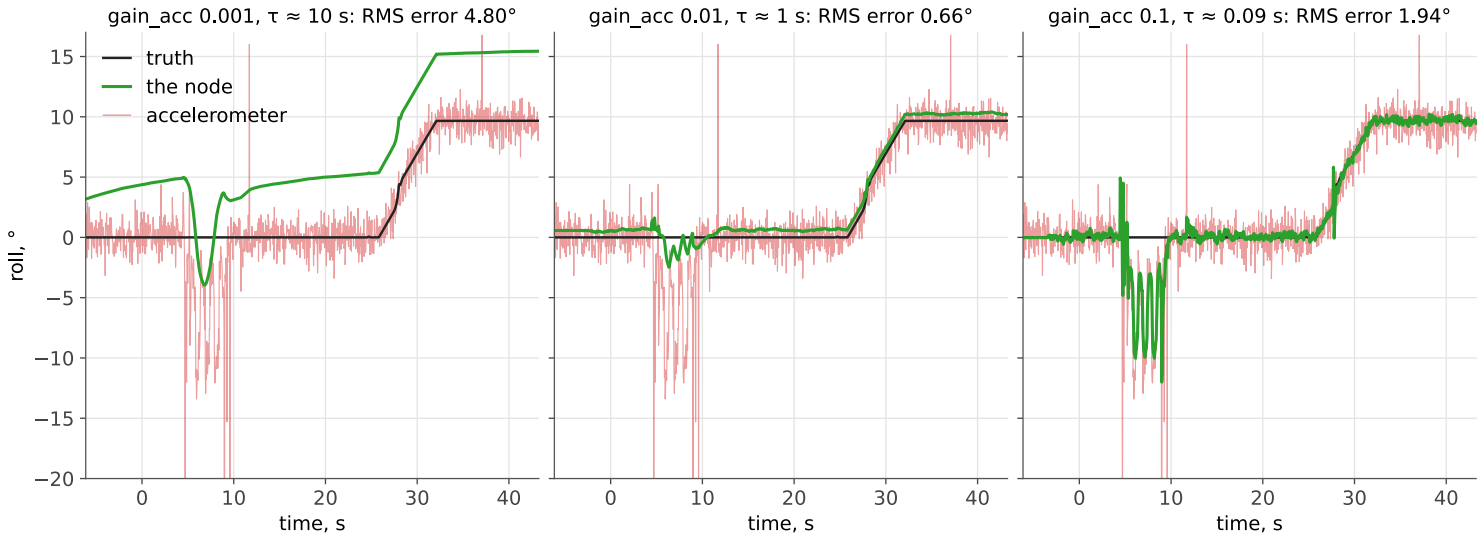
```cpp
// Prediction.
getPrediction(wx, wy, wz, dt, q0_pred, q1_pred, q2_pred, q3_pred);

// Correction (from acc):
// q_ = q_pred * [(1-gain) * qI + gain * dq_acc]
getAccCorrection(ax, ay, az, q0_pred, q1_pred, q2_pred, q3_pred,
                 dq0_acc, dq1_acc, dq2_acc, dq3_acc);
gain = gain_acc_;
scaleQuaternion(gain, dq0_acc, dq1_acc, dq2_acc, dq3_acc);
quaternionMultiplication(q0_pred, q1_pred, q2_pred, q3_pred, dq0_acc,
                         dq1_acc, dq2_acc, dq3_acc, q0_, q1_, q2_, q3_);
```

Parameters: `gain_acc`, `do_adaptive_gain: false`, `do_bias_estimation: false`, `use_mag: false`

🎯 github.com/CCNYRoboticsLab/imu_tools

PICKING THE TIME CONSTANT ON OUR ROBOT: THREE RUNS



$$\hat{\phi}_k = \alpha\,\phi^{-} + (1-\alpha)\,\phi_{acc} \qquad \texttt{gain_acc} = 1-\alpha \qquad \tau \approx \frac{\Delta t}{\texttt{gain_acc}}$$

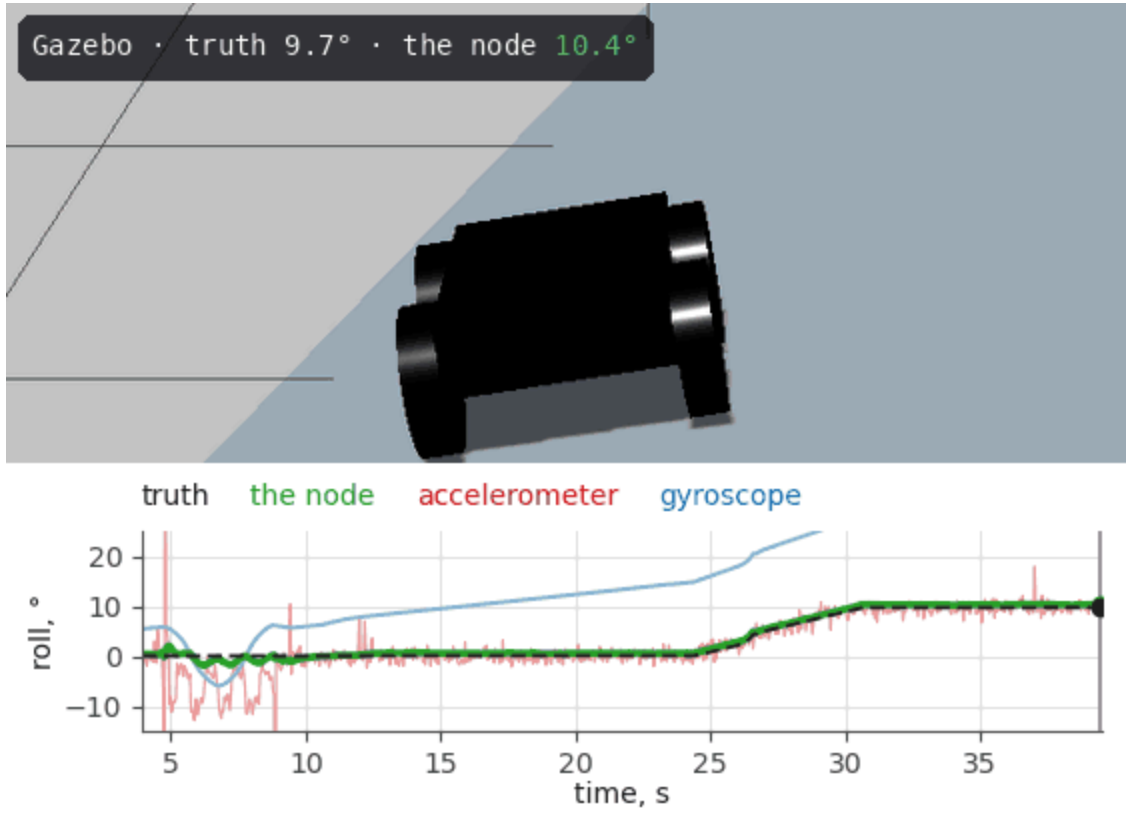
- `complementary_filter_node` on our robot, three runs: `gain_acc` = 0.001, 0.01, 0.1, so $\tau \approx 10$ s, 1 s, 0.09 s; `gain_acc` is ours, set once in the node's parameters
- $\tau \approx 10$ s: smooth, but it keeps the drift · $\tau \approx 0.09$ s: quick, but it believes the false roll in the turn · $\tau \approx 1$ s, the default: the smallest error

LEGEND

green line — $\hat{\phi}_k$, the node's estimate; k — the step number
light red line — ϕ_{acc}, the accelerometer's roll
black line — the truth
ϕ^{-} — the gyroscope's prediction
α — the prediction's weight
`gain_acc` — imu_tools' name for the accelerometer's share, $1-\alpha$
τ — the time constant, s; Δt — one IMU step, 0.01 s
RMS error — the root mean square of the node's roll minus the true roll, over the run: its typical error

🎯 github.com/CCNYRoboticsLab/imu_tools

IMU_TOOLS ON OUR ROBOT: ROLL HOLDS, YAW DRIFTS



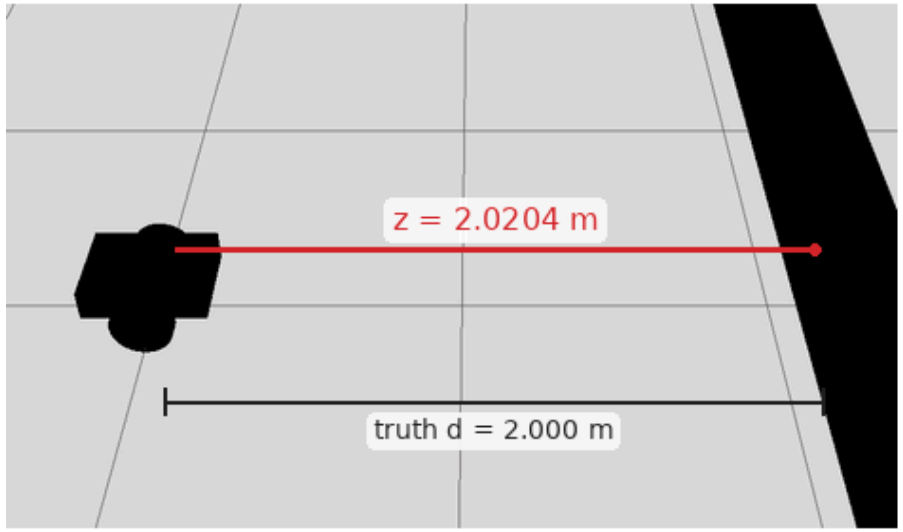
- `/imu/data_raw` → `complementary_filter_node` → `/imu/data`: the same readings, plus an orientation
- `gain_acc` = 0.01, the default: the roll is 0.6° off on average, 2° at worst — in the turn
- standing on the slope it stays 0.6° off: each step the bias b adds $b\,\Delta t$, the accelerometer takes 1 % back; they balance near $b\,\tau \approx 0.6°/s \times 1\ s$
- **yaw still drifts**: gravity says nothing about a turn around the vertical; later: the lidar fixes it
- we picked the weight by trying. **Is there a best weight — and what if the robot moves?**

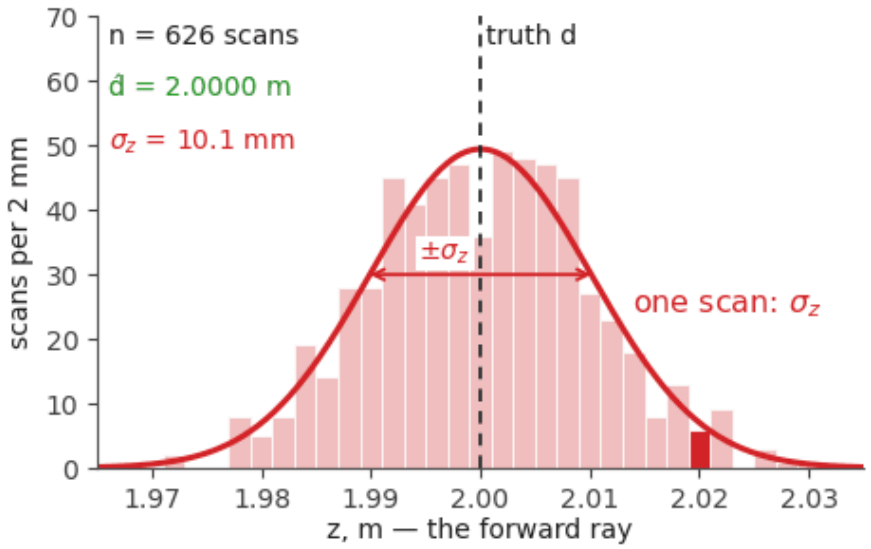
LEGEND

dashed line — the true roll, from Gazebo
green line — the node's roll
faint red — the accelerometer;
faint blue — the gyroscope
grey vertical line — the moment Gazebo shows
b — the gyroscope's bias: the same small error in every reading
Δt — one IMU step, 0.01 s
τ — the time constant: about 1 s at `gain_acc` 0.01
yaw — the turn around the vertical

🎯 github.com/CCNYRoboticsLab/imu_tools

A BELIEF IS A GUESS AND ITS DOUBT





$$\mathcal{N}(\hat{d},\ \sigma^2_{\hat{d}})$$

A belief is a best guess $\hat{d}$ and its typical error $\sigma_{\hat{d}}$: the Gaussian says how likely each distance is.

The red Gaussian is where one scan lands: after one scan $\hat{d} = z, \sigma_{\hat{d}} = \sigma_z$.

LEGEND

$\mathcal{N}(\hat{d}, \sigma^2_{\hat{d}})$ — a normal distribution, a Gaussian: centre $\hat{d}$, variance $\sigma^2_{\hat{d}}$

$\hat{d}$ — the best guess of the distance, m; the hat — an estimate

$\sigma_{\hat{d}}$ — its typical error: the Gaussian's width; σ^2 — a variance

z — one scan of the lidar's forward ray, m

σ_z — how far one scan scatters: 10.1 mm, measured

red bars — the scans, 2 mm bins, the latest bright; **red curve** — where one scan lands

green number — $\hat{d}$, the mean so far; dashed — the truth

probability — the area under the curve; ±σ holds 68 %

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/03-Gaussians.ipynb

AVERAGING SHRINKS THE ERROR

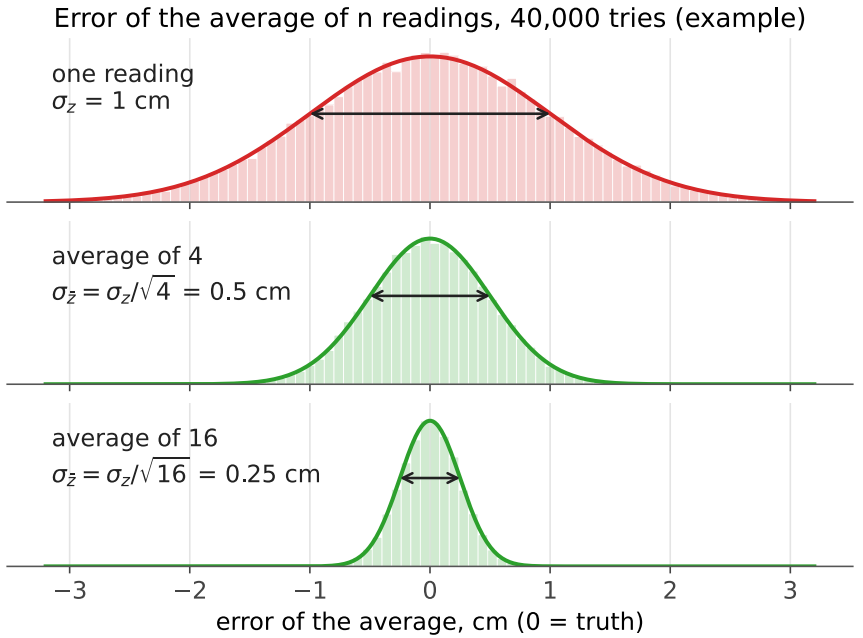
Each of 4 readings is off by +1 or −1 cm, equally likely (example): σ_z = 1 cm

SUM OF ERRORS	WAYS	ERROR OF THE AVERAGE	SQUARED
±4	1 + 1	±1	1
±2	4 + 4	±0.5	0.25
0	6	0	0

the mean square: **0.25** cm², so $\sigma_{\bar{z}}$ = **0.5 cm**

$$\sigma_{\bar{z}} = \frac{\sigma_z}{\sqrt{n}}$$

How many readings to average: 4 halve the error, 16 quarter it.



LEGEND

σ_z — the error of one reading (one scan)

n — the number of readings averaged

$\bar{z}$ — their average; a bar means an average

$\sigma_{\bar{z}}$ — the error of their average: the root of the mean square

16 = 2^4 sign combinations of four readings; 1-4-6-4-1 — how many of them give each sum

red — one reading; **green** — the average of 4 and of 16: bars — 40,000 tries, curves — the Gaussian of the formula, arrows — ±σ

youtu.be/7mYDHbrLEQo?t=141

THE RECURSIVE AVERAGE KEEPS NO OLD SCANS

$$\hat{d}_n = \hat{d}_{n-1} + \frac{1}{n}\left(z_n - \hat{d}_{n-1}\right)$$

The recursive average: the same average, keeping no old scans.

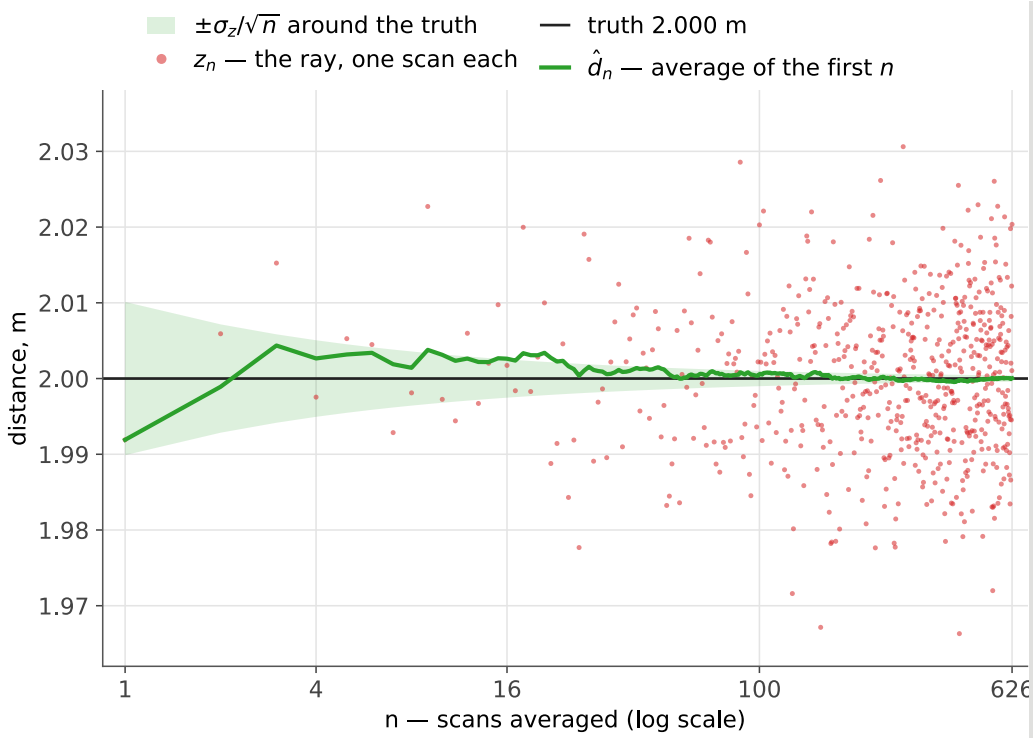
Four scans (example):

n	z_n, M	THE SHARE $\frac{1}{n}$	$\hat{d}_n$, M
1	2.04	1	2.04
2	2.02	½	2.03
3	2.03	⅓	2.03
4	2.07	¼	2.04

Python · pandas — a table library: the mean of all scans so far

```
d_hat = z.expanding().mean()    # running average
```

📖 pandas.pydata.org/docs/reference/api/pandas.Series.expanding.html



LEGEND

n — the number of the scan

z_n — the n-th scan of the lidar's forward ray, m

$\hat{d}_n$ — the average of the first n scans; $\hat{d}_{n-1}$ — before the new one

$\frac{1}{n}$ — the share of the new scan

$z_n - \hat{d}_{n-1}$ — how much the new scan disagrees

same as $(z_1 + \cdots + z_n)/n$: multiply both sides by n

`z` — all scans in a pandas column (`Series`), one per row

`expanding()` — windows that grow by one scan: z_1; z_1, z_2; z_1, z_2, z_3; …

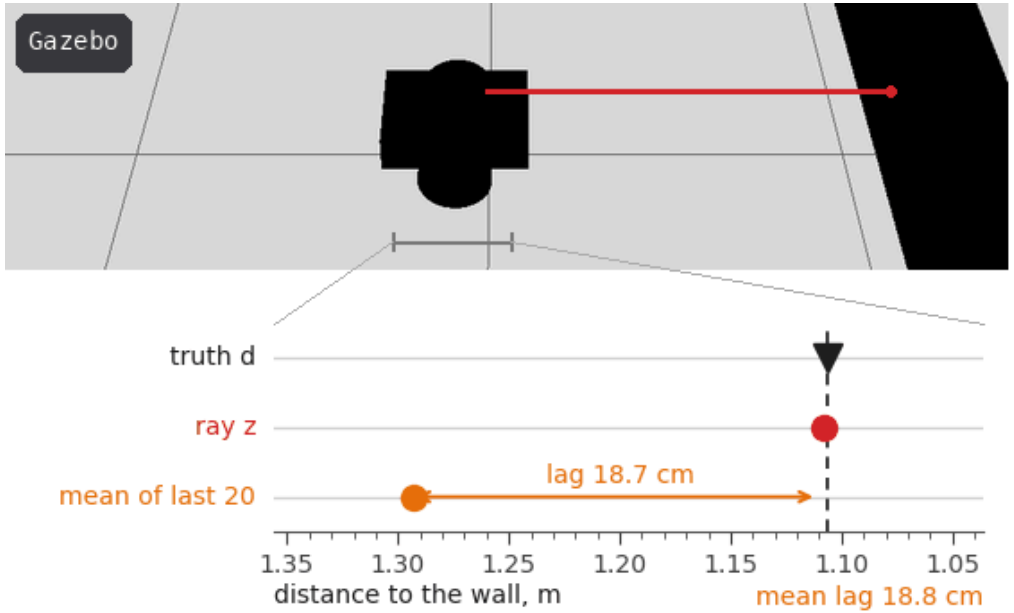
`.mean()` — each window's average: `d_hat` = $\hat{d}_1, \hat{d}_2, \ldots$

red dots — the scans; **green line** — their average; black line — the truth, 2.000 m

light band — ±σ_z/√n: the average's expected error; σ_z — one scan's error, 10.1 mm

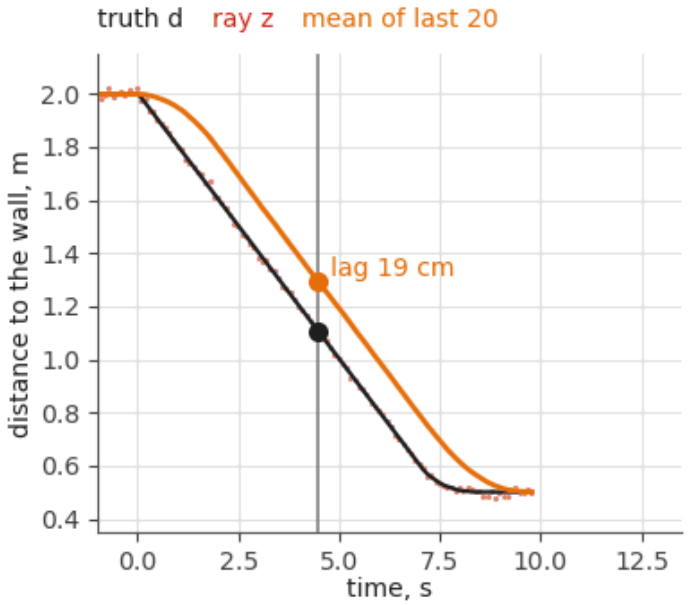
THEN AVERAGE THE LAST 20 SCANS WHILE DRIVING?

ANSWER: NO — A MOVING AVERAGE IS LATE



$$\text{lag} = \frac{n-1}{2}\, v\, \Delta t = 9.5 \cdot 0.2\ \text{m/s} \cdot 0.1\ \text{s} = 19.0\ \text{cm}$$

How far the average of the last n scans stays behind a moving robot; measured: 18.8 cm.



Python · pandas — a table library: the mean of the last 20 scans

```
d_avg = z.rolling(20).mean()
```

LEGEND

lag — how far the average is behind the truth, cm

n — scans in the window, 20

v — speed toward the wall, m/s

Δt — time between scans, 0.1 s

d — the true distance to the wall, z — the lidar's forward ray, m

$\frac{n-1}{2}$ — the window's middle lags this many scans: the mean of 0, 1, …, $n-1$

`z` — all scans in a pandas column (`Series`), one per row

`rolling(20)` — a window of the last 20 scans that slides by one: $z_1 \dots z_{20}$; $z_2 \dots z_{21}$; …

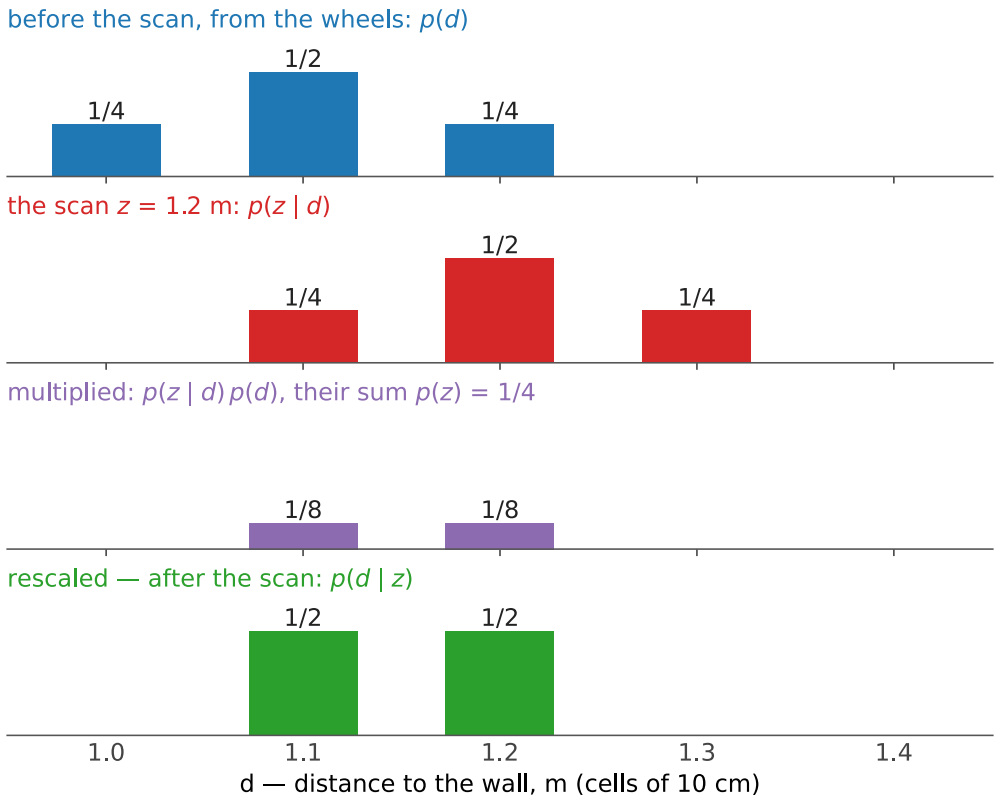
`.mean()` — each window's average: `d_avg`, the orange line

black — the truth · **red** — the ray · **orange** — the mean of the last 20 scans

grey vertical line — the moment the scene shows; dots — the truth and the mean at it

📖 pandas.pydata.org/docs/reference/api/pandas.Series.rolling.html

BAYES' RULE: THE BELIEF AFTER A SCAN



Our robot before the wall, five cells of 10 cm (example):

- **The robot knows** its belief before the scan — the wheel odometry says about 1.1 m; how its lidar errs — by a cell at most; and the scan — the forward ray reads 1.2 m
- **It does not know** the true distance d
- **It computes** the belief after the scan, for every cell

$$p(d \mid z) = \frac{p(z \mid d)\, p(d)}{p(z)}$$

Bayes' rule: from "how likely this scan is at each distance" to "how likely each distance is, given this scan".

LEGEND

d — a cell: a distance to the wall, m

z — the scan: what the lidar's forward ray reads, m

$p(\cdot)$ — a probability, 0 to 1; the bar $\mid$ reads «given»

$p(d)$ — the belief before the scan (prior): **blue bars**

$p(z \mid d)$ — the likelihood: how likely the ray reads z if the wall is at d: **red bars**

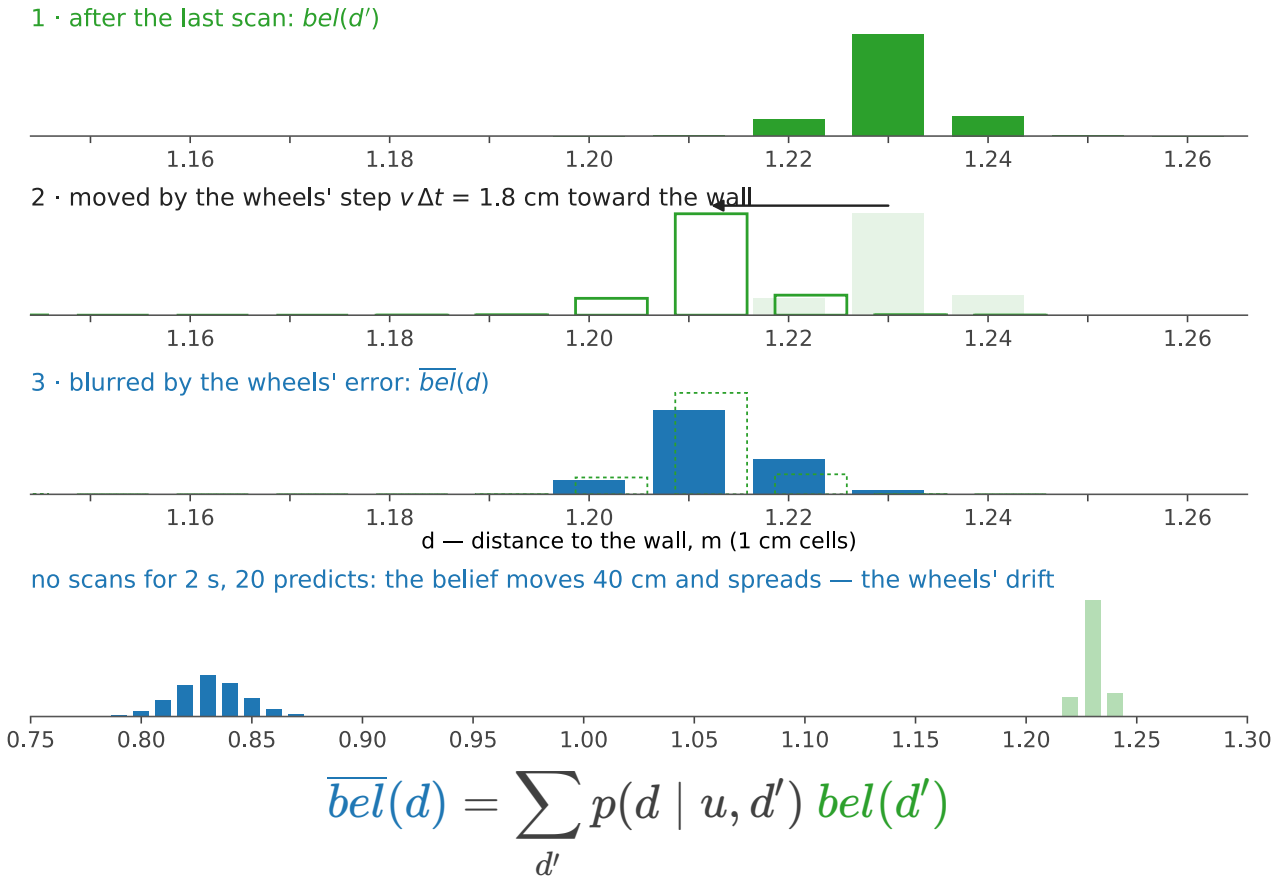
purple bars — the products, cell by cell

$p(z)$ — their sum; divided by it, the bars add up to 1

$p(d \mid z)$ — the belief after the scan (posterior): **green bars**

www.3blue1brown.com/lessons/bayes-theorem

PREDICT ON OUR GRID: MOVE THE BELIEF AND BLUR IT



$$\overline{bel}(d) = \sum_{d'} p(d \mid u, d')\, bel(d')$$

Predict: each cell collects what the motion brings into it.

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/02-Discrete-Bayes.ipynb

LEGEND

d — a cell after the motion: a distance to the wall, m

d' — a cell before it; the sum runs over all of them

u — the motion: the wheels' step $v\,\Delta t$, here 1.8 cm

$bel(d')$ — the belief after the last scan: **green bars**

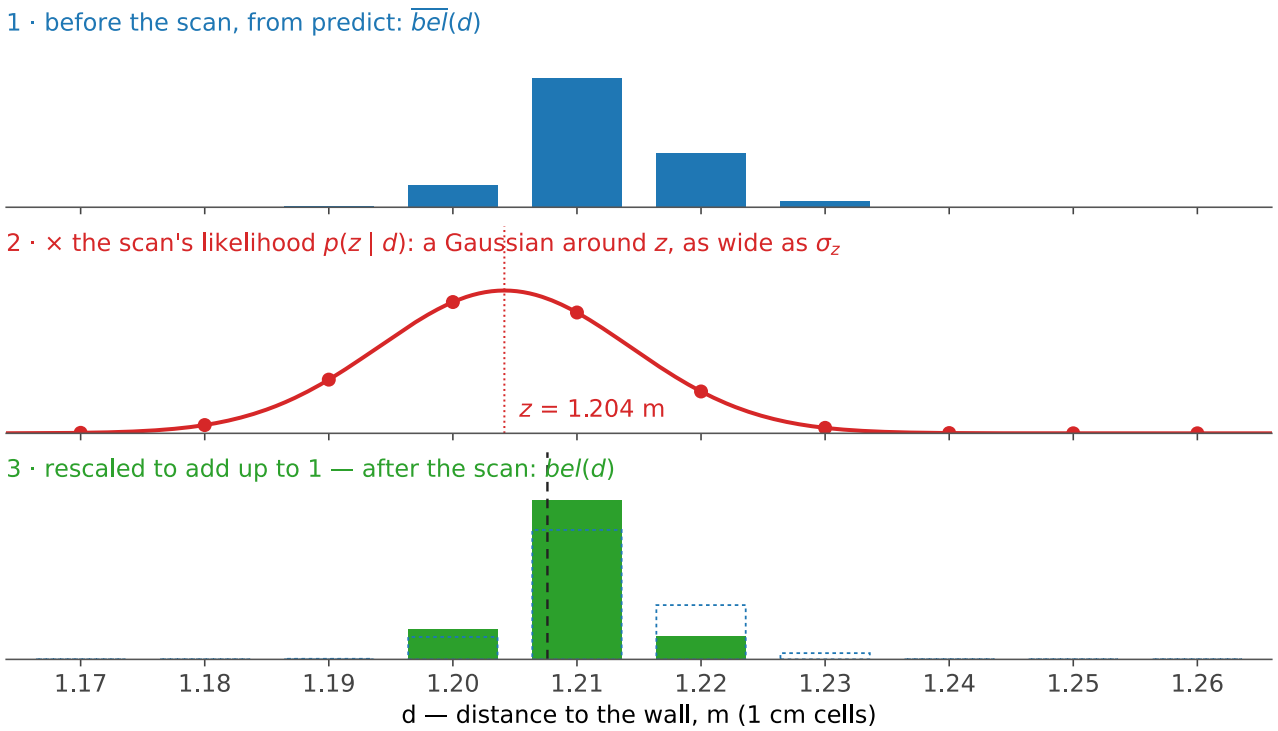
$p(d \mid u, d')$ — the chance that the step u takes the robot from d' to d: the wheels err

$\overline{bel}(d)$ — the belief before the next scan: **blue bars**; the bar on top — before the scan

outlined bars — the green bars, moved by u: each blue bar collects from those around it

the belief — one number per cell; the robot keeps nothing else

UPDATE ON OUR GRID: MULTIPLY BY THE SCAN'S LIKELIHOOD



$$bel(d) = \eta\, p(z \mid d)\, \overline{bel}(d)$$

LEGEND

d — a cell: a distance to the wall, m

z — the scan: what the lidar's forward ray reads, m

$\overline{bel}(d)$ — the belief before the scan, from predict: **blue bars**

$p(z \mid d)$ — the likelihood: how likely the scan z is if the wall is at d; a Gaussian around z, as wide as the lidar's error σ_z: **red curve**

$bel(d)$ — the belief after the scan: **green bars**; outlined — the blue bars, to compare

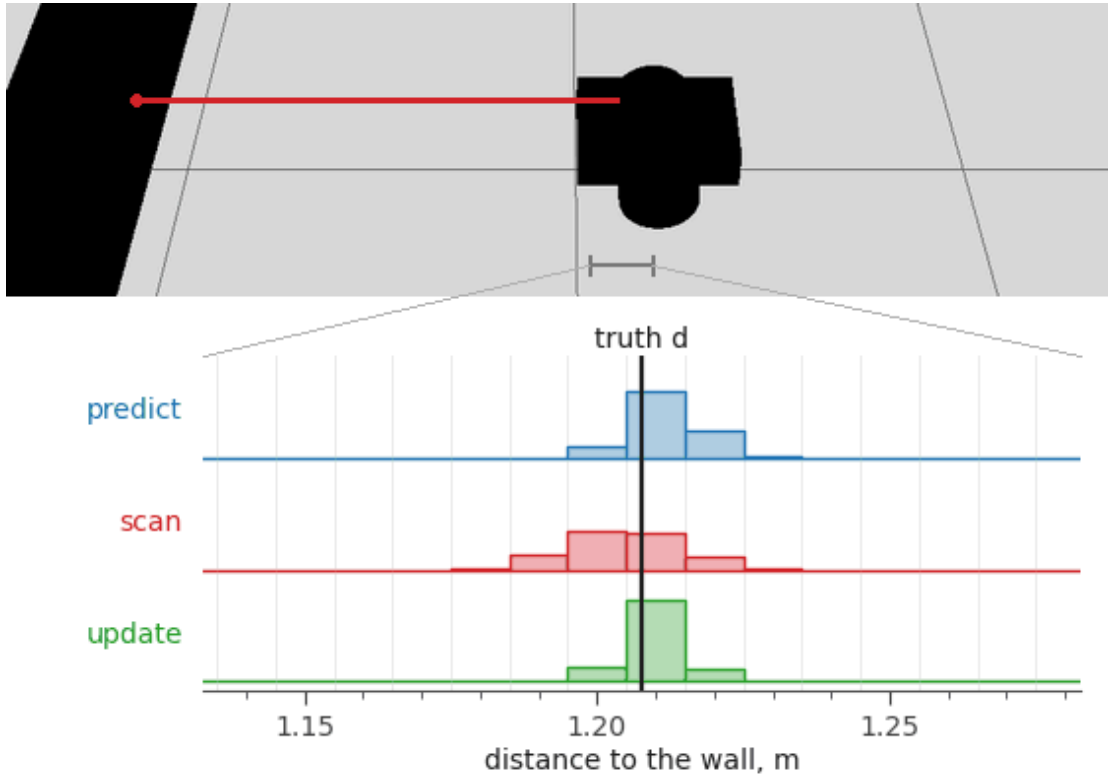
η — 1 / the sum of the products, so the bars add up to 1: the $1/p(z)$ of Bayes' rule

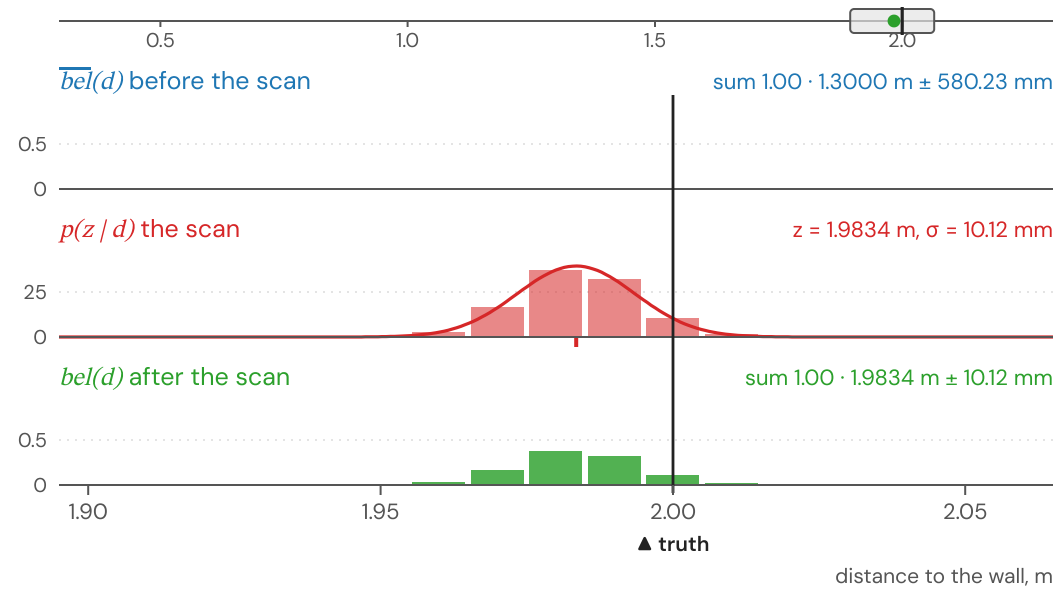
dashed line — the truth

Update: Bayes' rule on every cell — multiply by how well the cell explains the scan, then rescale.

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/02-Discrete-Bayes.ipynb

THE RECIPE ON A GRID: PREDICT, UPDATE, REPEAT





predict | update | ↺ reset | scan 9: still | scan 49 | scan 93: stop

scan 0 of 0–144

t = −0.92 s · update: z = 1.9834 m · after the scan 1.9834 m ± 10.12 mm · truth 2.0000 m

predict — move and blur: $\overline{bel}(d) = \sum_{d'} p(d \mid u, d')\, bel(d')$ · **scan** — its likelihood $p(z \mid d)$ · **update** — multiply, rescale: $bel(d) = \eta\, p(z \mid d)\, \overline{bel}(d)$ · repeat

d, d' — a cell now and before the motion · u — the motion · z — the scan · η — 1 / the sum · σ in the rows — a typical error · t — time from the start

FORMULA ↔ FILTERPY.DISCRETE_BAYES

$$\overline{bel}(d) = \sum_{d'} p(d \mid u, d')\, bel(d')$$

- `np.roll` / `shift` — move every cell by the motion u (`offset`, in cells)
- `convolve(…, kernel)` — the formula's sum: each cell collects its neighbours' belief with the weights `kernel = [0.1, 0.8, 0.1]` (example) — a blur by the wheels' error
- `mode='wrap'` — the cells close into a ring; `'constant'` — a wall: our drive; `cval` — what fills the cells that come in from beyond the wall: 0

$$bel(d) = \eta\, p(z \mid d)\, \overline{bel}(d)$$

- `prior` * `likelihood` — cell by cell
- `normalize` — divide by the sum: this is η; `np.asarray(…, dtype=float)` — the belief as an array of fractions

Python · filterpy/discrete_bayes/discrete_bayes.py — filterpy's grid Bayes filter, library code

```python
def predict(pdf, offset, kernel, mode='wrap', cval=0.):
    if mode == 'wrap':
        return convolve(np.roll(pdf, offset), kernel, mode='wrap')

    return convolve(shift(pdf, offset, cval=cval), kernel,
                    cval=cval, mode='constant')

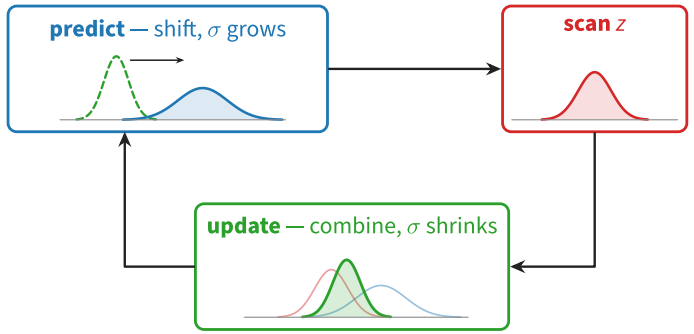
def update(likelihood, prior):
    posterior = prior * likelihood
    return normalize(posterior)

def normalize(pdf):
    pdf /= sum(np.asarray(pdf, dtype=float))
    return pdf
```

- d, d' — a cell now and before the motion; u — the motion; `pdf` — the belief, one number per cell
- $\overline{bel}, bel$ — the belief before and after the scan: `prior`, `posterior`
- $p(z \mid d)$ — the scan's likelihood over the cells: `likelihood`; η — 1 / sum

🎯 github.com/rlabbe/filterpy
▶ colab.research.google.com/github/StanfordASL/pora-exercises/blob/main/ch11/discrete_bayes.ipynb

THE KALMAN FILTER: **PREDICT**, THEN **UPDATE**



predict — the wheel odometry moves the guess; the doubt grows

$$d^- = \hat{d} - v\,\Delta t \qquad \sigma_{d^-}^2 = \sigma_{\hat{d}}^2 + \sigma_u^2$$

update — the scan pulls the guess; the doubt shrinks

$$K = \frac{\sigma_{d^-}^2}{\sigma_{d^-}^2 + \sigma_z^2}$$

$$\hat{d} = d^- + K\,(z - d^-) \qquad \sigma_{\hat{d}}^2 = (1 - K)\,\sigma_{d^-}^2$$

The grid's recipe for one Gaussian: two numbers per step — the guess and its doubt.

	BAYES FILTER ON A GRID	KALMAN FILTER
the belief	one number per cell: 201 numbers	one Gaussian: $\hat{d}$ and $\sigma_{\hat{d}}$
it needs	nothing: any shape, even two peaks	a belief that stays one Gaussian

youtu.be/ul3u2yLPwU0

LEGEND

each Gaussian — where the robot may be: the likeliest distance at its centre, less likely to the sides

$\hat{d}$ — the estimate of the distance to the wall, the green centre; the hat — an estimate; dashed green — the last one

$\sigma_{\hat{d}}$ — its typical error, the width; σ^2 — its variance, the doubt

d^-, σ_{d^-} — the prediction and its error; the minus — before the scan

$v\,\Delta t$ — one step by the wheel odometry: speed × time

σ_u — the error one step adds

z — the scan of the lidar's forward ray; σ_z — its error: the lidar's own, constant

K — the Kalman gain: the scan's weight, from 0 to 1

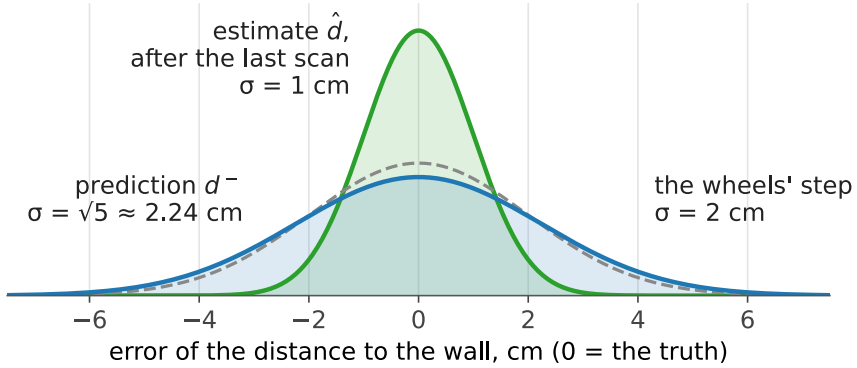
$z - d^-$ — the innovation: how far the scan is from the prediction

KALMAN **PREDICT**: THE DOUBT GROWS

The estimate is off by ±1 cm; one step of the wheels adds ±2 cm (example):

ESTIMATE	STEP	PREDICTION	SQUARED
+1	+2	+3	9
+1	−2	−1	1
−1	+2	+1	1
−1	−2	−3	9

mean square **5 = 1 + 4**: the variances add



$$d^- = \hat{d} - v\,\Delta t \qquad \sigma_{d^-}^2 = \sigma_{\hat{d}}^2 + \sigma_u^2$$

The Gaussian moves with the robot, its σ grows: variances add.

Python · filterpy/stats/stats.py — filterpy's helper that adds two Gaussians, library code

```
def add(mean1, var1, mean2, var2):
    return (mean1+mean2, var1+var2)
```

Only a measurement shrinks the doubt — the update, next.

youtu.be/d_qvLDhkg00

LEGEND

$\hat{d}$ — the estimate of the distance to the wall after the last scan, before the robot moves; $\sigma_{\hat{d}}$ — its typical error, 1 cm (example)

$v\,\Delta t$ — the step: how far the wheel odometry says the robot moved between two scans

σ_u — the error one step adds: the wheels', 2 cm (example); σ_u^2 — `Q` in filterpy and robot_localization

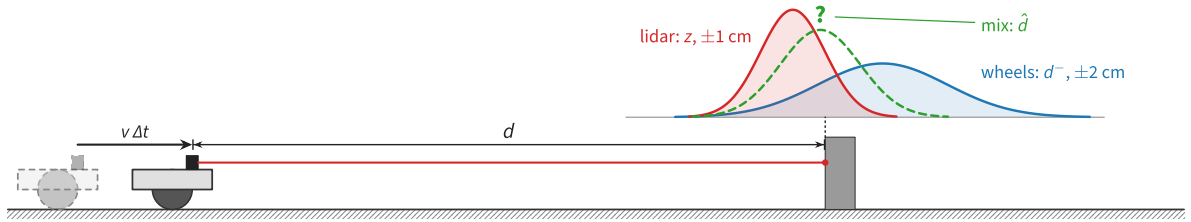
d^-, σ_{d^-} — the prediction and its error; the minus — before the scan

σ^2 — a variance; variances add, σ do not: 1 + 4 = 5, so σ_{d^-} = √5, not 1 + 2

`mean1`, `var1` — the estimate: $\hat{d}$, $\sigma_{\hat{d}}^2$; `mean2`, `var2` — the step: $-v\,\Delta t$, σ_u^2

green — the estimate; dashed — the step's error; blue — the prediction

WHEELS AND LIDAR: WHICH WEIGHT IS BEST?



- **The wheels predict:** $d^- = \hat{d}_{k-1} - v\,\Delta t$, off by σ_{d^-} = 2 cm (example)
- **The lidar measures** with its forward ray, one of 360, straight at the wall: $z = d + \text{noise}$, off by σ_z = 1 cm (example)
- **Mix them** with a weight w for the scan:

$$\hat{d} = (1 - w)\, d^- + w\, z = d^- + w\,(z - d^-)$$

The same weighted mix as in the complementary filter: which w is best?

	COMPLEMENTARY FILTER	KALMAN FILTER
the weight	fixed, picked by eye: 0.02	computed at every step from the two variances
the doubt	not kept	kept as σ^2: how sure the estimate is

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/04-One-Dimensional-Kalman-Filters.ipynb

LEGEND

d — the true distance to the wall

d^- — the prediction: the last estimate moved by the wheel odometry's step; the minus — before the scan

σ_{d^-} — the prediction's typical error

$\hat{d}_{k-1}$ — the estimate one step earlier, at step $k-1$

z — the lidar's forward ray; σ_z — its typical error

$\hat{d}$ — the mix, the fused estimate: the new estimate

$v\,\Delta t$ — how far the robot moved in one step: speed × time

w — the scan's weight: any number from 0 to 1. The loop's K is the best w — found next

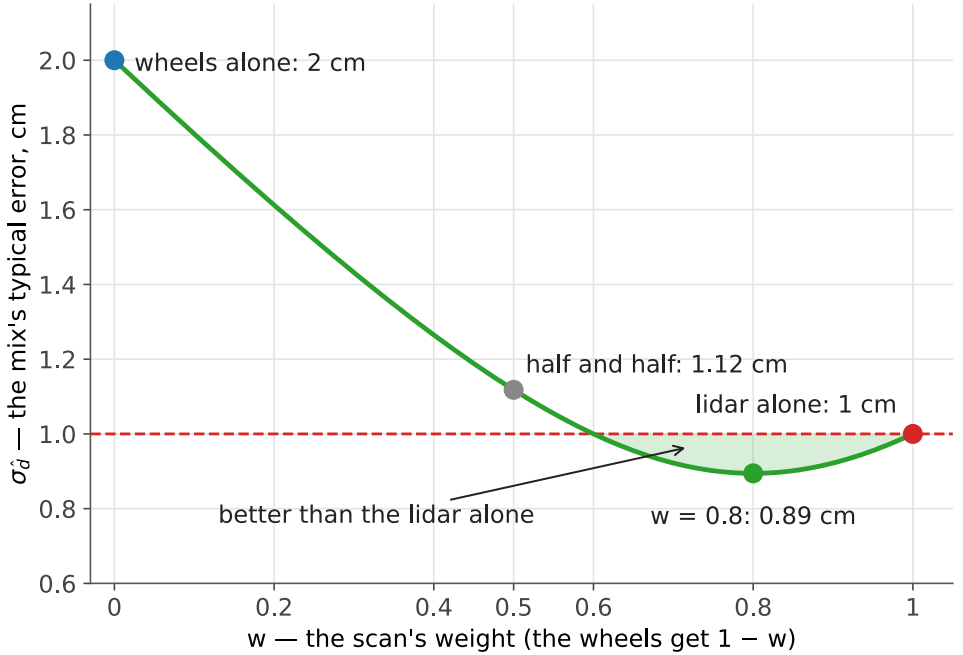
blue, **red** — where the wall may be, by the wheel odometry and by the lidar: likeliest at the centre; dotted line — where it truly is; **dashed green** — the mix, not known yet

THE RIGHT WEIGHT BEATS BOTH SENSORS

Each error is + or −, equally likely: lidar ±1 cm, wheels ±2 cm (example). Mix with $w = 0.8$:

LIDAR	WHEELS	MIX	SQUARED
+1	+2	1.2	1.44
+1	−2	0.4	0.16
−1	+2	−0.4	0.16
−1	−2	−1.2	1.44

mean square **0.8** cm² — less than the lidar's 1 cm²



LEGEND

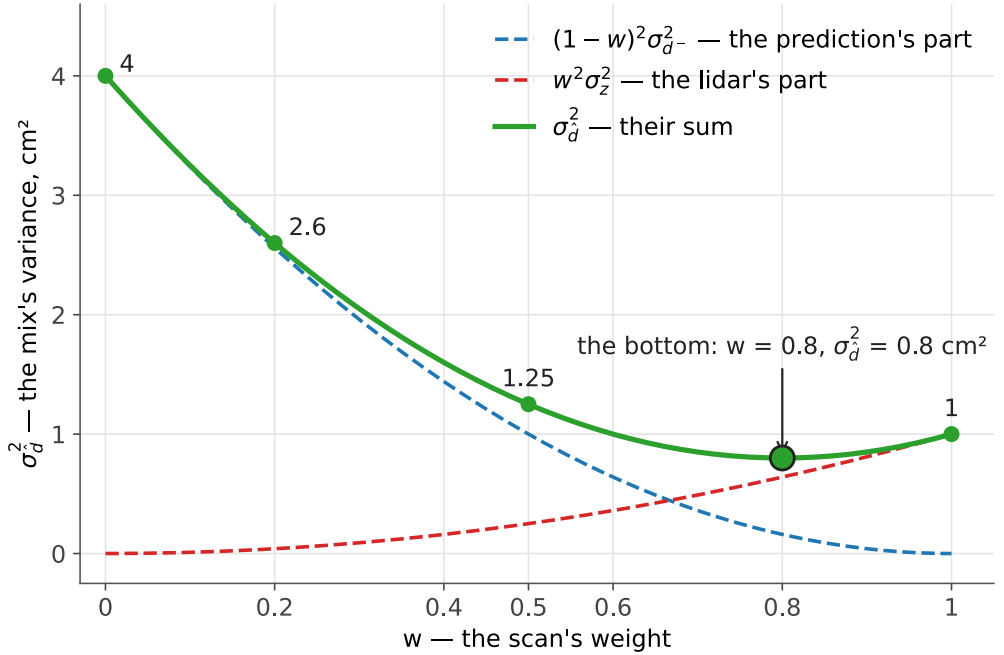
w — the scan's weight: here 0.8 for the lidar, 0.2 for the wheels
lidar, wheels — the error of each, cm; every sign equally likely
mix — 0.8 × the lidar's error + 0.2 × the wheels' error
mean square — the average of the error²
$\sigma_{\hat{d}}$ — the mix's typical error: the root of the mean square
lidar alone: σ_z = 1 cm · wheels alone: σ_{d^-} = 2 cm (example)
green curve — $\sigma_{\hat{d}}$ for every weight; **red dashed** — the lidar alone; shaded — where the mix beats it

THE ERROR OF THE MIX IS A PARABOLA

$$\sigma^2_{\hat{d}}(w) = (1-w)^2\,\sigma^2_{d^-} + w^2\,\sigma^2_z$$

The error of the mix for every weight: the sum of the two parts has a bottom.

$\sigma^2_{d^-} = 4, \sigma^2_z = 1$ cm² (example)



LEGEND

$\sigma^2_{\hat{d}}(w)$ — the mix's variance, its doubt: the average of its error² over many scans, cm²

w — the scan's weight, from 0 to 1

$\sigma^2_{d^-}$ — the prediction's variance: the average of its error²

σ^2_z — the lidar's variance

$(1-w)^2\sigma^2_{d^-}$ — the prediction's part: its error × (1 - w), so its variance × (1 - w)²: **blue dashed**

$w^2\sigma^2_z$ — the lidar's part: its error × w, its variance × w^2: **red dashed**

green — their sum, $\sigma^2_{\hat{d}}$; dots — a few weights, with their values

youtu.be/ul3u2yLPwU0

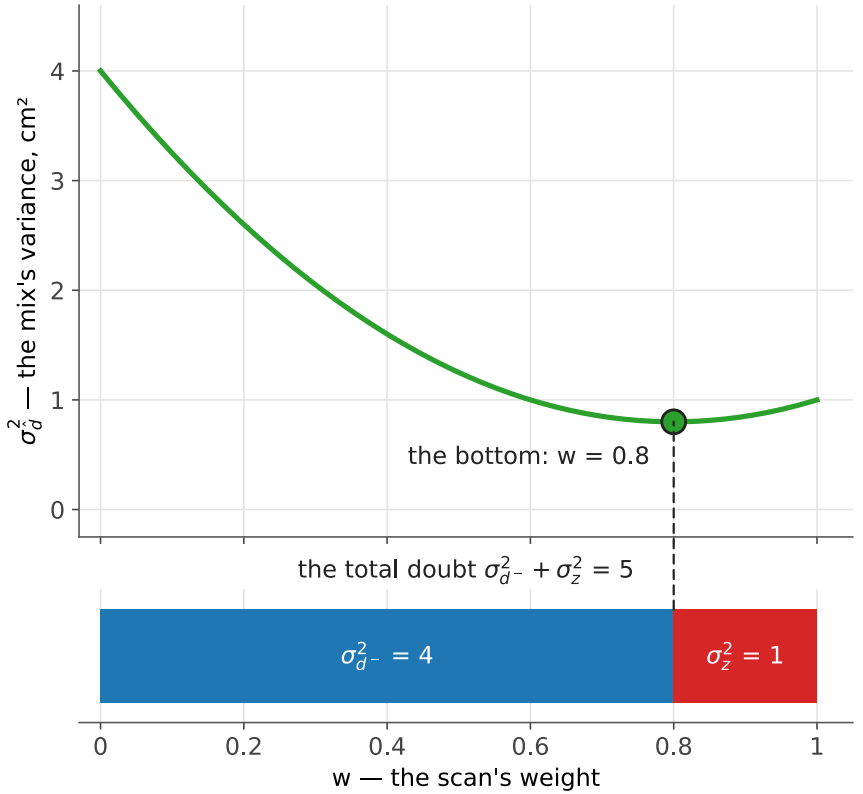
THE BOTTOM OF THE PARABOLA IS THE KALMAN GAIN K

$$K = \frac{\sigma_{d^-}^2}{\sigma_{d^-}^2 + \sigma_z^2}$$

The best weight: the prediction's share of the total doubt.

Three pairs of noises, cm² (example); the picture is the first:

CASE	$\sigma_{d^-}^2$	σ_z^2	K
wheels unsure	4	1	$4/(4+1)$ = **0.8**
lidar unsure	1	4	$1/(1+4)$ = **0.2**
equally sure	1	1	$1/(1+1)$ = **0.5**



LEGEND

green curve — the mix's variance $\sigma_{\hat{d}}^2$ for every weight w of the scan; **green dot** — its bottom, the best w

dashed line — from the bottom down to the bar: the bar splits right under it

bar — the total doubt $\sigma_{d^-}^2 + \sigma_z^2$: **blue** — the prediction's part, **red** — the lidar's

K — the Kalman gain: the best weight of the scan, from 0 to 1

$\sigma_{d^-}^2$ — the prediction's variance: its typical error, squared

σ_z^2 — the lidar's variance

$\sigma_{\hat{d}}^2$ — the mix's variance, its doubt, cm²

$\sigma_{d^-} = 0 \to K = 0$: keep the prediction · $\sigma_z = 0 \to K = 1$: take the scan

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/04-One-Dimensional-Kalman-Filters.ipynb

FROM ONE NUMBER TO MATRICES

	OUR DRIVE: ONE NUMBER	THE MATRIX FORM: MANY NUMBERS
the state	d	a column of n numbers: the pose x, y, θ
its doubt	$\sigma^2_{d^-}$	P^-, $n \times n$: each number's variance on the diagonal; off it, how two errors depend on each other — **not diagonal**
what the sensor reads	$z = d$, the distance itself	$z = H \cdot$state: H, $m \times n$, **picks** from the state what the sensor reads
the sensor's noise	σ^2_z	R, $m \times m$: **diagonal** when each reading errs on its own
a doubt through a number	$a\,\sigma^2 a = a^2\sigma^2$	$H\,P^-H^\top$: H on both sides
divide by S	$1/S$	multiply by S^{-1}, the inverse matrix
the gain K	one number: the scan's weight	$n \times m$: a reading moves every number whose error depends on it

H picks from the state what the sensor reads: *z* = *H*·state

our drive
the lidar reads *d*

$$z = \boxed{1} \cdot \boxed{d} = d$$

the pose (x, y, θ)
the IMU reads the yaw

$$z = \underbrace{\boxed{0}\,\boxed{0}\,\boxed{1}}_{H,\ 1 \times 3} \cdot \underbrace{\begin{matrix}\boxed{x}\\ \boxed{y}\\ \boxed{\theta}\end{matrix}}_{\text{state}} = \theta$$

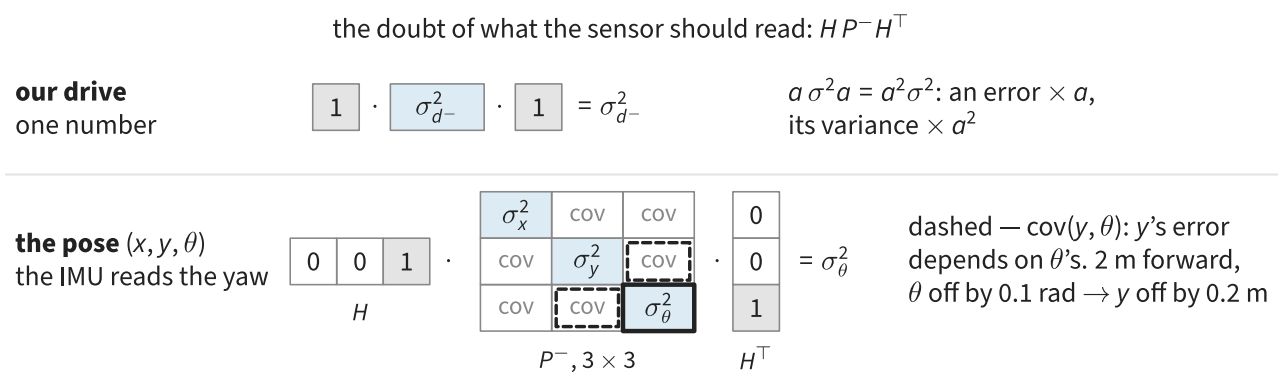
1 — the sensor reads this number, 0 — not

LEGEND

z — the reading: the lidar's distance, the IMU's yaw
H — the measurement matrix: picks from the state what the sensor reads; n — state numbers, m — readings
P^-, R — the prediction's and the sensor's covariance matrices
a — a number an error is multiplied by; σ^2 — that error's variance
$H^\top$ — H turned over; S — the reading's whole doubt; S^{-1} — the inverse of S; K — the Kalman gain
grey — H's ones; framed — what they pick

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/06-Multivariate-Kalman-Filters.ipynb

WHAT $H\,P^{-}H^{\top}$ COMPUTES



- $H\,P^{-}H^{\top}$ **— the doubt of the reading we expect:** the prediction's doubt, carried into what the sensor reads
- H **twice:** the error passes through H, and a variance is a square — so H on both sides, as a in $a\,\sigma^2 a$
- **Who fills P^{-}:** the filter — predict ties the errors, update shrinks them; we set only the start and the noises

LEGEND

H — picks from the state what the sensor reads; $H^{\top}$ — the same ones as a column

P^{-} — the prediction's covariance matrix: variances on the diagonal, blue

cov — a covariance: how two errors depend on each other

σ^2_x, σ^2_y, σ^2_θ — the variances of x, y and the yaw θ; $\sigma^2_{d^-}$ — our prediction's

a — a number an error is multiplied by; σ^2 — its variance

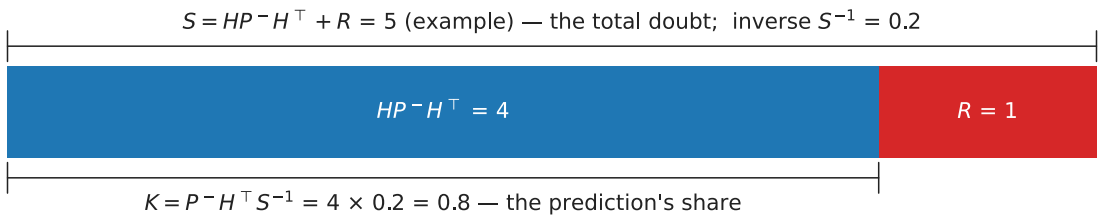
framed — what the ones pick; dashed — cov(y, θ)

www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/

THE KALMAN GAIN IN MATRICES, PIECE BY PIECE

The matrix form, as filterpy writes it: $K = P^{-}H^{\top}(HP^{-}H^{\top} + R)^{-1}$ — our 1D gain, piece by piece:

PIECE	OUR DRIVE, 1D	MATRICES	WHAT IT IS
1	$\sigma_{d^-}^2$	$H\,P^{-}\,H^{\top}$	the doubt of the reading we expect
2	$\sigma_{d^-}^2 + \sigma_z^2$	$S = HP^{-}H^{\top} + R$	plus the sensor's noise: variances add
3	$\sigma_{d^-}^2$ on top	$P^{-}H^{\top}$	H once: each state number's error paired with the reading's
4	divide by the sum	multiply by S^{-1}	the inverse instead of a division
K	$\dfrac{\sigma_{d^-}^2}{\sigma_{d^-}^2 + \sigma_z^2}$	$P^{-}H^{\top}S^{-1}$	how far each state number moves



With one number, every piece is the 1D one: K is the bottom of the parabola again.

LEGEND

K — the Kalman gain; H — picks what the sensor reads

P^{-}, R — the prediction's and the sensor's covariance matrices

S — the reading's whole doubt bar — S (example): **blue** — the prediction's part, **red** — the lidar's; under the blue part — K

LINEAR ALGEBRA

AB — each row of A times each column of B, summed; $(m \times n)(n \times p) = m \times p$

$A^{\top}$ — rows become columns

A^{-1} — the inverse: $A^{-1}A = I$, the identity; for a number, $1/A$

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/06-Multivariate-Kalman-Filters.ipynb