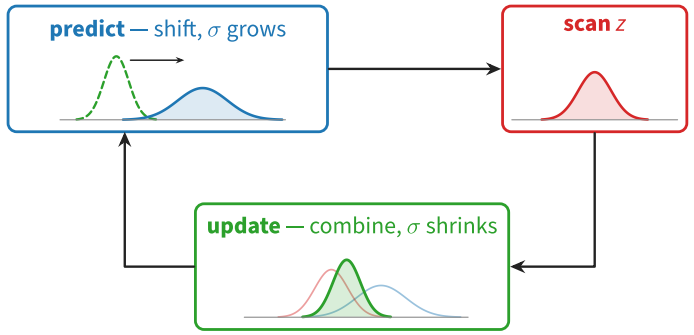
AUTONOMOUS
INTELLIGENCE GROUP

LECTURE 7 — STATE ESTIMATION AND FUSION (II)

ONE ESTIMATE FROM NOISY SENSORS: FROM KALMAN IN MATRICES TO THE EXTENDED KALMAN FILTER (EKF)

Topics: Kalman gain · Covariance · EKF · robot_localization

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

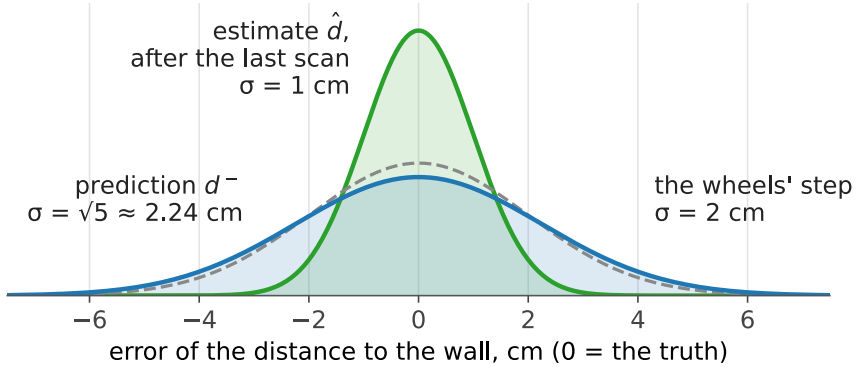
$z - d^-$ — the innovation: how far the scan is from the prediction

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

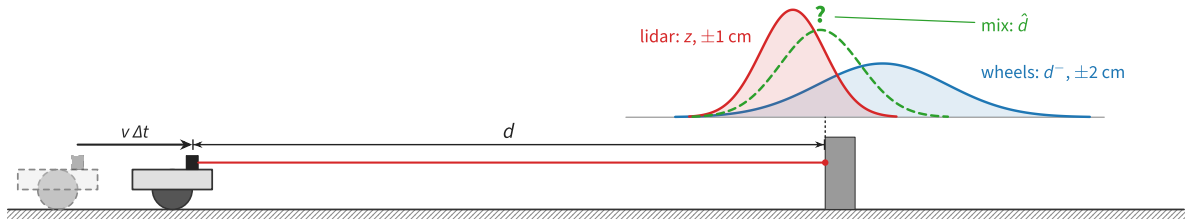
d^-, σ_{d^-} — the prediction and its error; the minus — before the scan

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

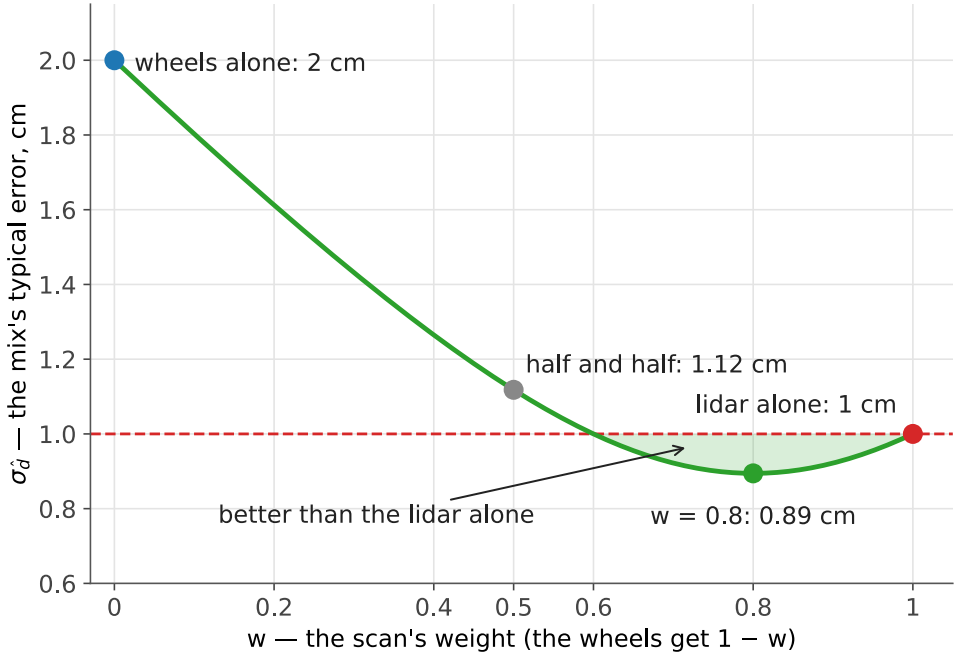
blue, **red** — where the wall may be, by the wheel odometry and by the lidar: likeliest at the centre; dotted line — where it truly is; **dashed green** — the mix, not known yet

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

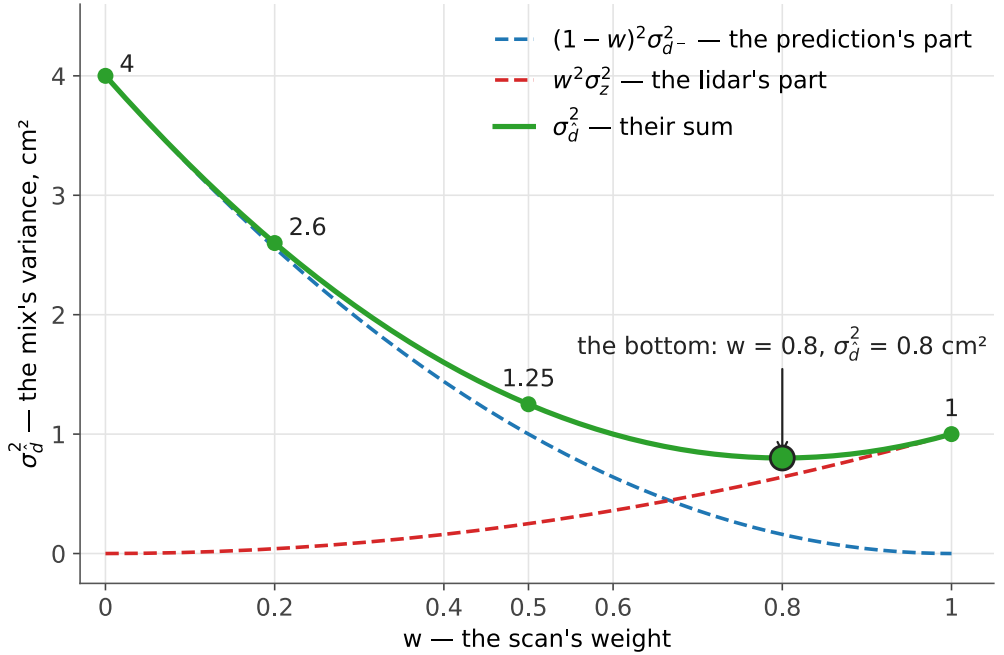
w — the scan's weight: here 0.8 for the lidar, 0.2 for the wheels
lidar, wheels — the error of each, cm; every sign equally likely
mix — 0.8 × the lidar's error + 0.2 × the wheels' error
mean square — the average of the error²
$\sigma_{\hat{d}}$ — the mix's typical error: the root of the mean square
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

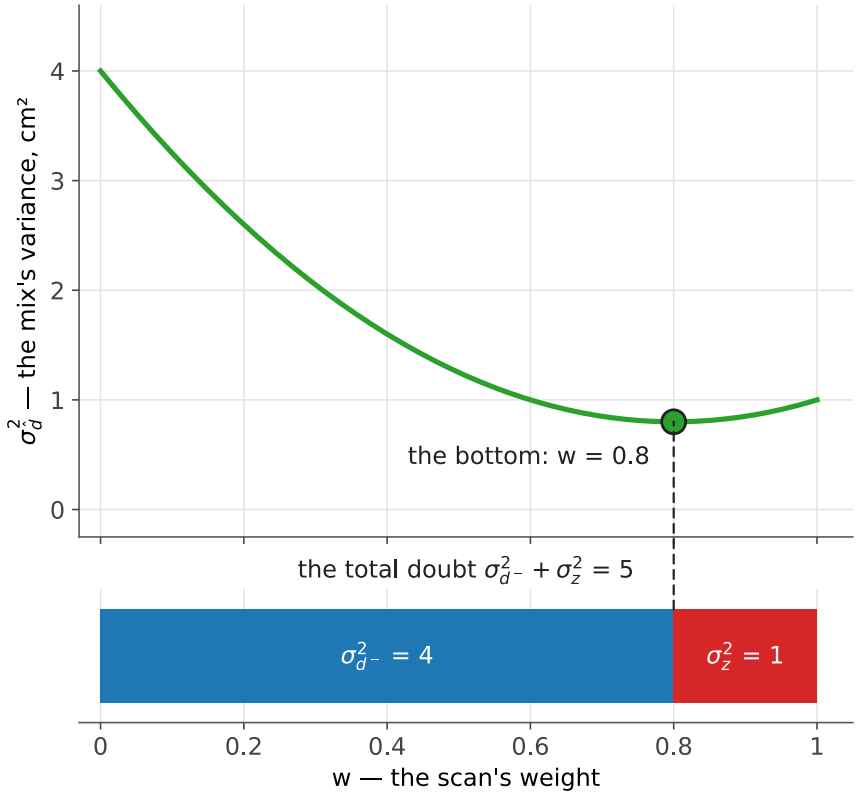
THE BOTTOM OF THE PARABOLA IS THE KALMAN GAIN K

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

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

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

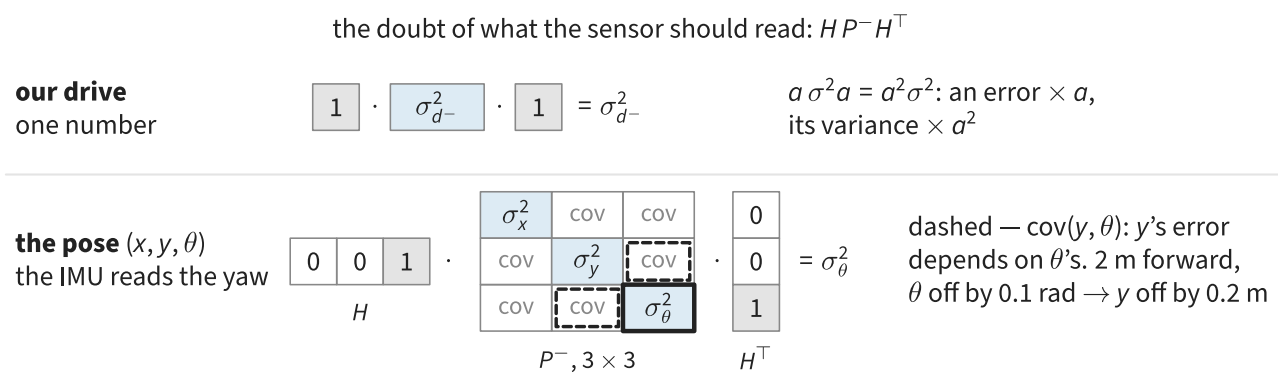
1 — the sensor reads this number, 0 — not

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; σ^2_{d-} — our prediction's

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

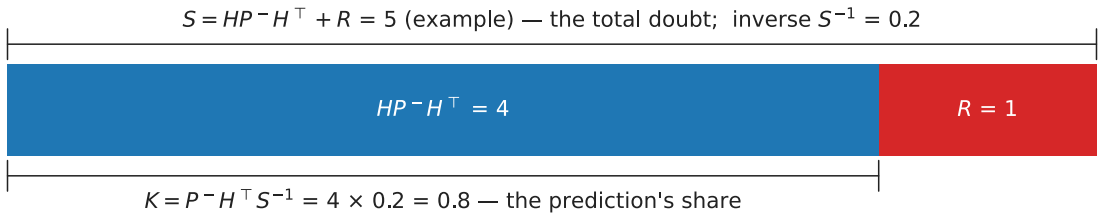
framed — what the ones pick; dashed — cov(y, θ)

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

FORMULA ↔ FILTERPY: THE KALMAN GAIN

$$K = P^{-} H^{\top} \left(H P^{-} H^{\top} + R \right)^{-1}$$

Python · filterpy/kalman/kalman_filter.py — filterpy's KalmanFilter.update, library code

```
# common subexpression for speed
PHT = dot(self.P, H.T)

# S = HPH' + R
# project system uncertainty into measurement space
self.S = dot(H, PHT) + R
self.SI = self.inv(self.S)
# K = PH'inv(S)
# map system uncertainty into kalman gain
self.K = dot(PHT, self.SI)
```

$P^{-} H^{\top}$

$S = H P^{-} H^{\top} + R$

S^{-1} — the inverse of S

$K = P^{-} H^{\top} S^{-1}$

The formula's pieces, line for line.

- S^{-1} — **the inverse of** S (`inv`): a matrix is not divided but multiplied by its inverse; for one number, $1/S$
- **We set** R and, for predict, $Q = \sigma_u^2$ · **the filter computes** P^{-}, S, K at every scan

LEGEND

blue — the prediction's part, red — the lidar's

K — the Kalman gain, the scan's weight: `self.K`

P^{-} — the prediction's doubt, our $\sigma_{d^-}^2$: `self.P`

R — the lidar's doubt, our σ_z^2: `R`

H — picks what the sensor reads: 1, the distance itself

$H^{\top}$ — H turned over: `H.T`

`PHT` — $P^{-} H^{\top}$, computed once for speed

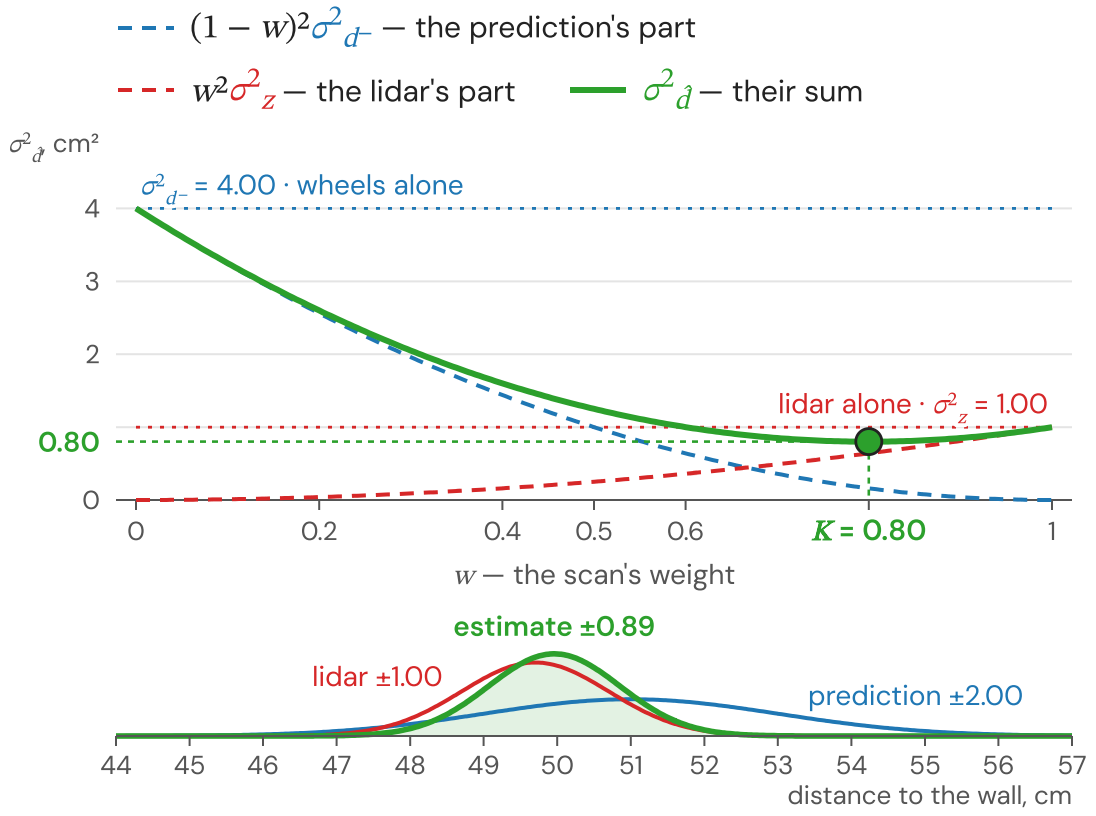
S — the whole doubt: `self.S`; `SI` — S^{-1}

`dot` — the matrix product: rows of the left times columns of the right

🎯 github.com/rlabbe/filterpy

MOVE THE NOISES: K FOLLOWS THE MORE ACCURATE SENSOR

The filter computes, every scan: $\sigma^2_{d^-} = \sigma^2_{\hat{d}} + \sigma^2_u$ · **we set once:** σ_u and σ_z · here both doubts are sliders: what if the prediction were less or more sure



the prediction by the wheel odometry — the filter computes it

σ_{d^-} 2.00 cm

$\sigma^2_{d^-} = 2.00^2 = 4.00$ cm²

the lidar — we set it once

σ_z 1.00 cm

$\sigma^2_z = 1.00^2 = 1.00$ cm²

the best weight: $K = \sigma^2_{d^-}/(\sigma^2_{d^-} + \sigma^2_z)$
$= 4.00/(4.00 + 1.00) = 0.80$

$\sigma^2_{\hat{d}}$ = the bottom = $(1-K)\sigma^2_{d^-}$
$= (1 - 0.80) \cdot 4.00 = 0.80$ cm²

below both $\sigma^2_{d^-}$ and σ^2_z: the estimate is more accurate than either sensor

reset

LEGEND

σ_{d^-} — the prediction's doubt as a typical error: the upper slider

$\sigma_{\hat{d}}$ — the estimate's typical error after the last scan

σ_u — what one step of the wheel odometry adds: its speed error × the time between scans

σ_z — the lidar's typical error, measured standing: the lower slider

w — the scan's weight; K — the best w, the bottom of the curve

$\sigma^2_{\hat{d}}(w)$ — the estimate's variance for each w

start (example): σ_{d^-} = 2 cm, σ_z = 1 cm

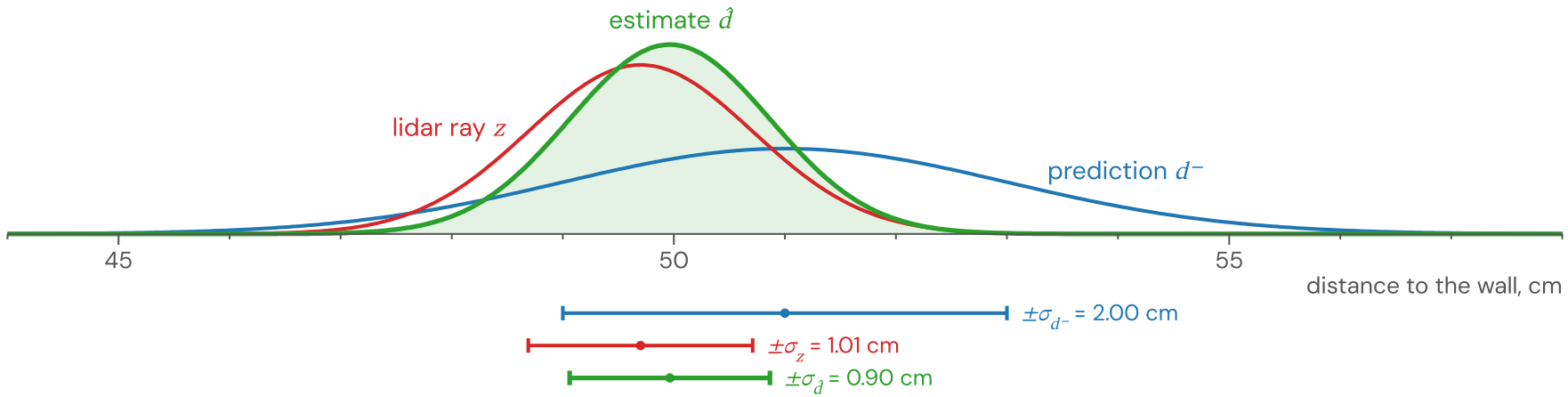
THE **UPDATE** COMBINES TWO GAUSSIANS

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

Python · filterpy/stats/stats.py · mul — filterpy's helper: the product of two Gaussians, library code

```
mean = (var1*mean2 + var2*mean1) / (var1 + var2)
var = 1 / (1/var1 + 1/var2)
```

As on the grid, the update multiplies the belief by the scan's Gaussian.



prediction

d^- 51.0 cm

σ_{d^-} 2.0 cm

lidar ray

z 49.7 cm

σ_z 1.0 cm

reset

estimate: $\hat{d}$ = 49.96 cm, $\sigma_{\hat{d}}$ = 0.90 cm

LEGEND

blue — the prediction, red — the forward ray, green — the estimate

$\hat{d}$, $\sigma_{\hat{d}}$ — the estimate after the update, its typical error

d^-, σ_{d^-} — the prediction, its typical error: the filter computes both in predict

z — the ray's reading at this scan; σ_z — its typical error, from the datasheet or measured standing: we set it once

K — the Kalman gain, the scan's best weight

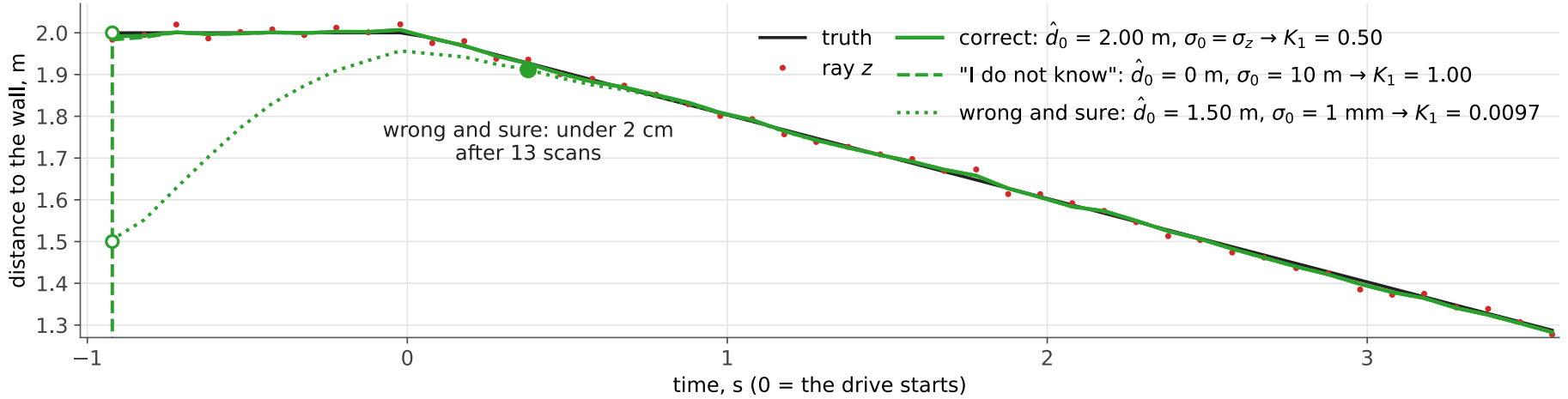
σ^2 — a variance; ±σ holds 68 %

start: 51 ± 2 cm (example); the ray 49.7 ± 1.01 cm, measured

`mean1`, `var1` — d^-, $\sigma^2_{d^-}$; `mean2`, `var2` — z, σ^2_z; `mean`, `var` — $\hat{d}$, $\sigma^2_{\hat{d}}$

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

STARTING THE FILTER



$$K_1 = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_z^2}$$

The first scan's weight: the start's doubt stands where the prediction's was.

- **We set the start** before the first scan: a first guess $\hat{d}_0$ and σ_0 — how far $\hat{d}_0$ may be from the truth; in `robot_localization` — `initial_estimate_covariance`
- **σ_0 is the error of whatever gave $\hat{d}_0$:** a tape measure from the wall — its accuracy, about 1 cm; the first scan — σ_z; nothing known — so large that $\pm 3\sigma_0$ covers every place the robot can be
- **Unsure? Take it larger:** a large σ_0 costs one scan; a wrong small one — many

LEGEND

green lines — the filter's estimate from three starts: solid — correct, the true 2 m, $\sigma_0 = \sigma_z$; dashed — "I do not know", 0 m, σ_0 = 10 m; dotted — wrong and sure, 0.5 m short, σ_0 = 1 mm; hollow dots — the starts; the bold dot — within 2 cm
black — the truth, Gazebo's ground truth; red dots — the scans z of the forward ray
K_1 — the Kalman gain of the first scan
$\hat{d}_0$ — the first guess of the distance to the wall; σ_0 — its typical error: how far it may be from the truth
σ_z — the lidar's typical error, 10.1 mm on our drive

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

FORMULA ↔ FILTERPY: THE LOOP

predict $d^- = \hat{d} - v\,\Delta t \qquad \sigma^2_{d^-} = \sigma^2_{\hat{d}} + \sigma^2_u$

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

Python · filterpy/kalman/kalman_filter.py — filterpy's KalmanFilter.predict() and update(), library code

```
# predict
# x = Fx + Bu
self.x = dot(F, self.x) + dot(B, u)
# P = FPF' + Q
self.P = self._alpha_sq * dot(dot(F, self.P), F.T) + Q

# update: S, SI and K first, as in the Kalman gain's code
self.y = z - dot(H, self.x)
self.x = self.x + dot(self.K, self.y)
I_KH = self._I - dot(self.K, H)
self.P = dot(dot(I_KH, self.P), I_KH.T) + dot(dot(self.K, R), self.K.T)
```

- **We set** `Q` and `R` once · **the filter computes** `x`, `P` and K at every step

LEGEND

`Q` — σ^2_u, one odometry step's doubt

`R` — σ^2_z, the lidar's doubt

`x` — the state: d^-, then $\hat{d}$

`P` — its variance: $\sigma^2_{d^-}$, then $\sigma^2_{\hat{d}}$

`y` — the scan minus the prediction, $z - d^-$

K — the Kalman gain; `I_KH` — $1 - K$

F — the state's own change: 1 here

$B\,u$ — the step $-v\,\Delta t$: speed × time

H — picks what the sensor reads: 1

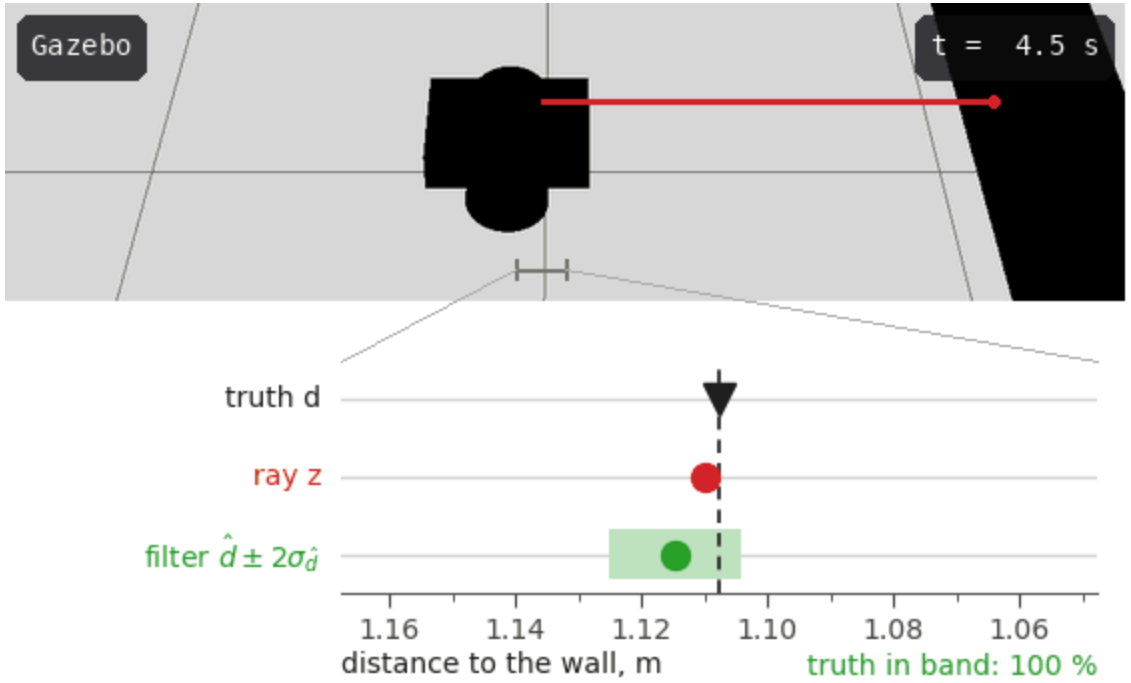
`dot` — the matrix product, rows × columns; for numbers, ×

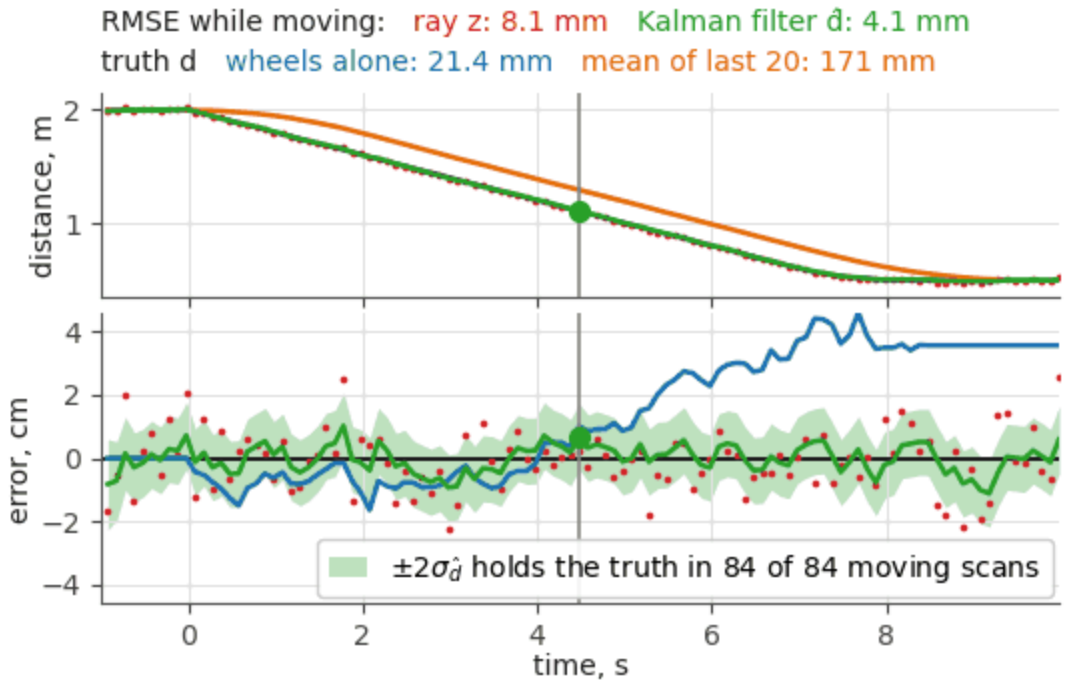
`.T` — turned over; `_alpha_sq` — 1

🎯 github.com/rlabbe/filterpy

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

THE FILTER ON OUR DRIVE: NO LAG, LESS NOISE



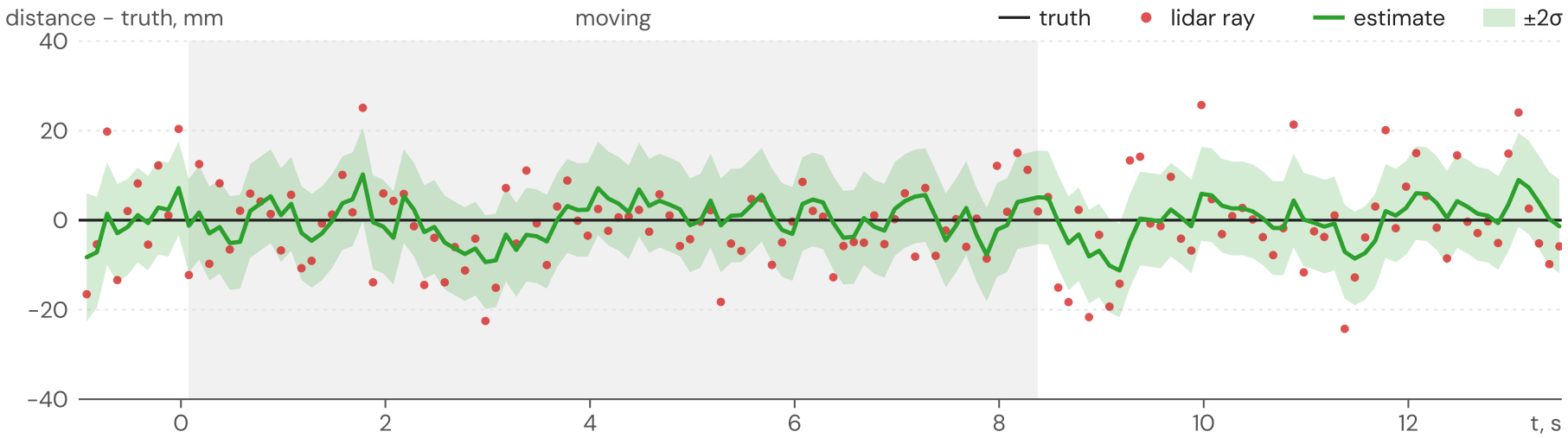


$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N} e_k^2}$$

- **RMSE** (root mean square error) — a line's typical error over the drive, in mm: one measure for the filter, the ray, the wheels and the average
- e_k — the line's error at scan k: the line minus the truth d; N — the 84 scans while moving · grey bracket — the 12 cm zoomed in under the robot
- **Conclusion:** the filter has the smallest error and no lag — it predicts the motion

TUNING: THE LIDAR'S NOISE AND THE WHEELS' NOISE

σ_z: measured standing $\sigma_u = \sigma_v \, \Delta t$



both noises as measured · K at the end 0.268 · while moving: RMSE 4.1 mm, truth inside ±2σ 100 % (84 of 84)

σ_u^2 × 1

reset

× 0.01 × 0.1 **× 1 (measured)** × 10 × 100

Conclusion: the measured noises give the smallest error

youtu.be/ul3u2yLPwU0

LEGEND

black — the truth, zero error; red dots — each scan's error; green — the estimate's error, its band $\pm 2\sigma_{\hat{d}}$

σ_z — the lidar's typical error; filterpy's `R` = σ_z^2

σ_u — one step's distance error, m: the speed error held for Δt; filterpy's `Q` = σ_u^2

σ_v — the wheels' speed error, m/s, from the wheels' driver

Δt — the time between two scans, s

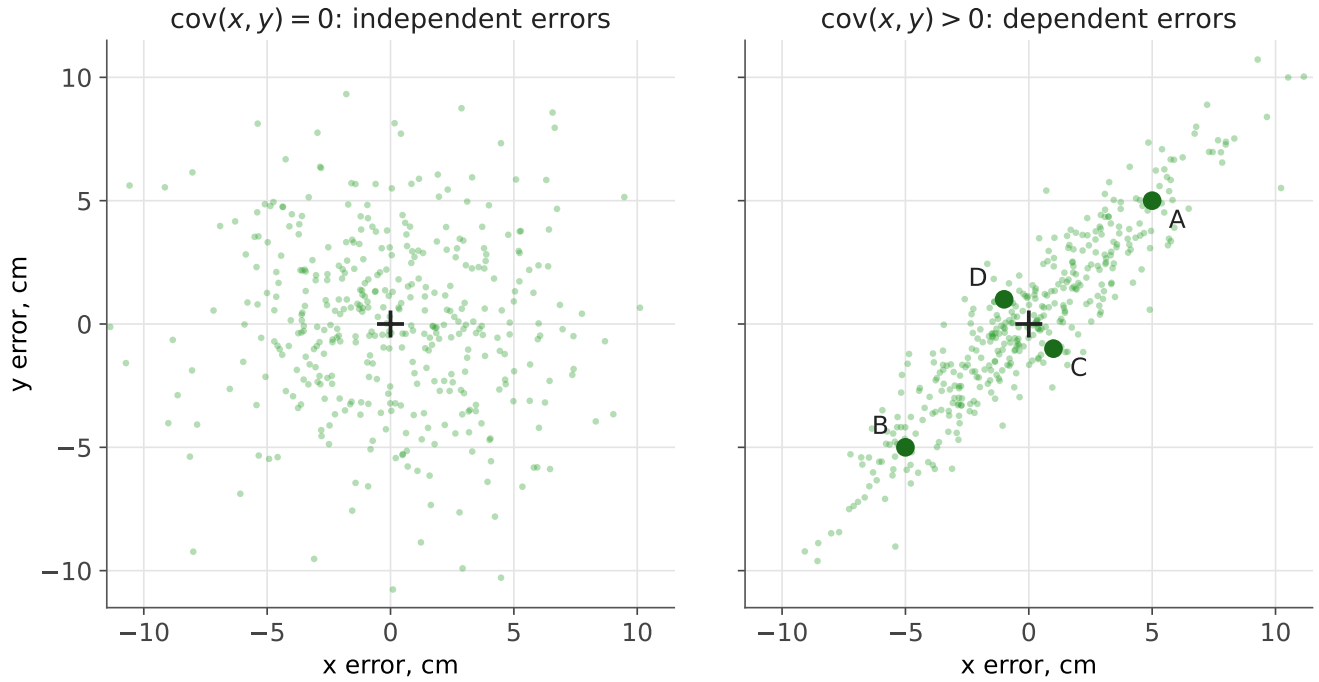
we set σ_z and σ_u once; the slider multiplies σ_u^2, σ_z stays as measured

K — the scan's weight; $\sigma_{\hat{d}}$ — the estimate's typical error

THE COVARIANCE MATRIX

$$P = \begin{bmatrix} \sigma_x^2 & \mathrm{cov}(x,y) \\ \mathrm{cov}(x,y) & \sigma_y^2 \end{bmatrix} \qquad \mathrm{cov}(x,y) = \text{the mean of } x \cdot y$$

Variances on the diagonal; off it, how the errors depend on each other.



LEGEND

P — the covariance matrix: the doubt of several numbers; 3 × 3 for the pose (x, y, θ), θ — the yaw; here only x, y

σ_x^2, σ_y^2 — the variance of each, cm²: on the diagonal, the same in both clouds

$\mathrm{cov}(x,y)$ — the covariance: multiply the two errors of each pair, average over many pairs, cm²; off the diagonal; 0 when the errors are independent

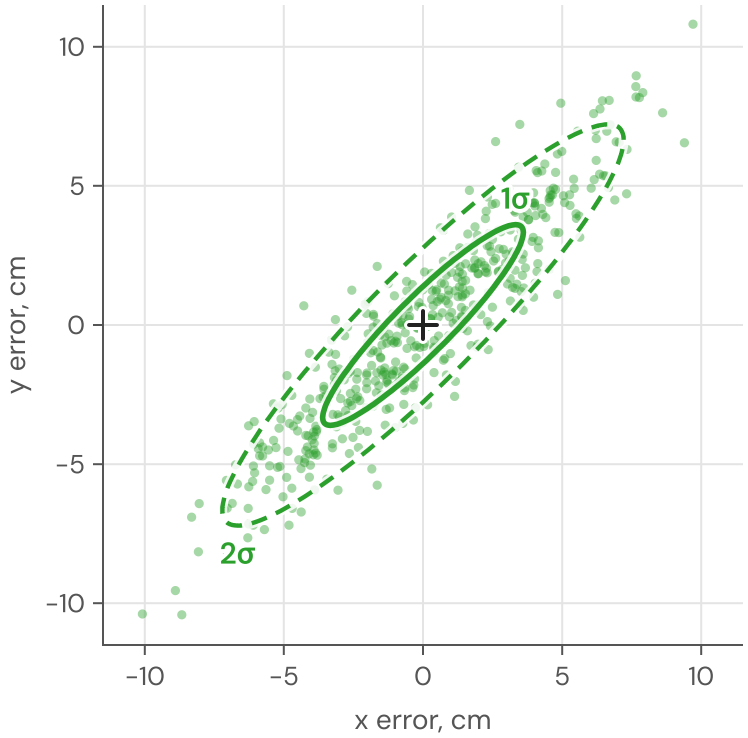
x, y — the position errors, cm

green dots — many position errors; black cross — zero error, the true position

A, B — both errors the same way: $x \cdot y > 0$; C, D — opposite ways: $x \cdot y < 0$

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/05-Multivariate-Gaussians.ipynb

THE COVARIANCE ELLIPSE: $\pm\sigma$ IN 2D



σ_x 3.61 cm

σ_y 3.61 cm

ρ 0.92

$$P = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{bmatrix}$$

— inside 1σ: **39 %** of the points

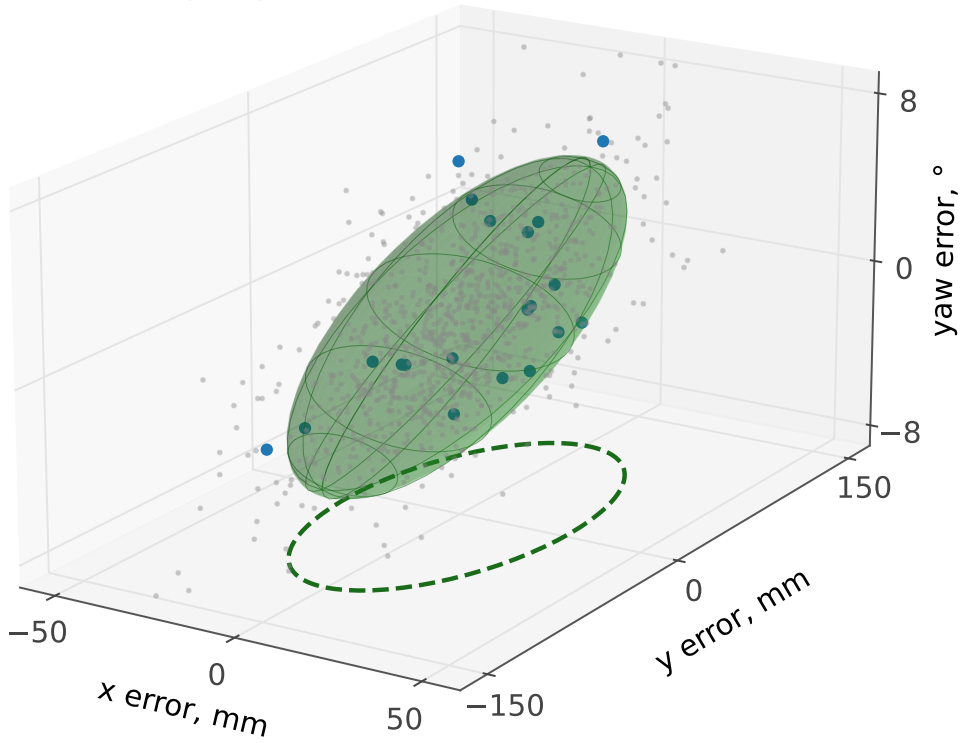
- - - inside 2σ: **86 %** of the points

reset

- $\rho = \mathrm{cov}(x, y) \,/\, (\sigma_x \sigma_y)$ — the correlation: the covariance scaled to -1 … 1, so its slider tilts the cloud and leaves the spread
- solid ellipse — 1σ, dashed — 2σ; σ_x, σ_y — the typical error along x, y; P — the covariance matrix; start — the slanted cloud (example)
- **Conclusion:** the 1σ ellipse holds fewer than half the points — RViz draws 1σ, so an honest filter is judged at 2σ

THREE VARIABLES: THE ELLIPSE BECOMES AN ELLIPSOID

- 1000 simulated drives
- 20 Gazebo runs
- the 2σ ellipsoid
- its shadow: the x–y ellipse



$$P = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & \mathrm{cov}(y,\theta) \\ 0 & \mathrm{cov}(y,\theta) & \sigma_\theta^2 \end{bmatrix}$$

- The pose has **three** numbers: P is 3 × 3, the ellipse becomes an **ellipsoid**
- It leans: the y error **depends on** the yaw error — a robot that faces a little left drives to the left
- The x error comes from the speed, not the yaw: zeros
- Its shadow on the floor is the x–y ellipse RViz draws

LEGEND

grey dots — 1000 simulated drives of 2 m with our wheels' noise

blue dots — 20 Gazebo runs: where the wheel odometry says each ended

green — their 2σ ellipsoid; dashed — its shadow, the 2σ ellipse of x and y

x — the error along the drive, y — sideways, mm; θ — the yaw error, °

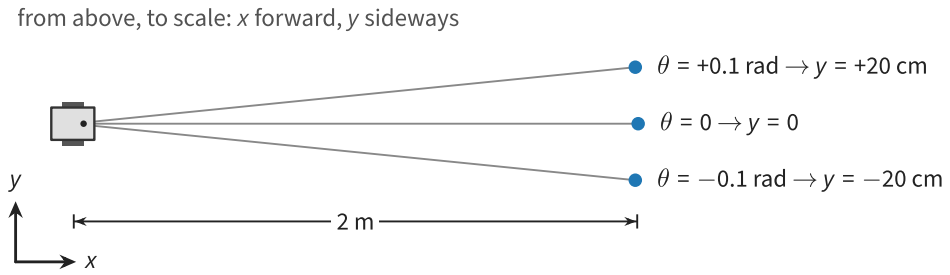
P — the covariance matrix of (x, y, θ)

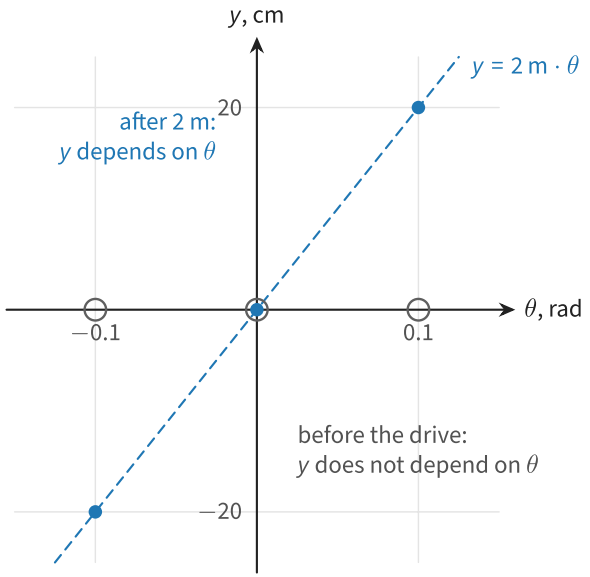
$\sigma_x^2, \sigma_y^2, \sigma_\theta^2$ — the variance of each error: the diagonal

$\mathrm{cov}(y,\theta)$ — the covariance of the y and yaw errors, > 0; zeros — x depends on neither (≈ 0 on these drives)

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/05-Multivariate-Gaussians.ipynb

A HEADING ERROR BECOMES A SIDEWAYS ERROR





$$\begin{aligned} y^- &= 1 \cdot y + 2 \cdot \theta \\ \theta^- &= 0 \cdot y + 1 \cdot \theta \end{aligned} \quad \Rightarrow \quad F = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \qquad P^- = F\,P\,F^\top : \begin{bmatrix} 0 & 0 \\ 0 & 0.01 \end{bmatrix} \rightarrow \begin{bmatrix} 0.04 & 0.02 \\ 0.02 & 0.01 \end{bmatrix}$$

The step turns the yaw's doubt into a sideways doubt and ties y to θ.

LEGEND

grey lines — copies of the robot driving 2 m, each with its own yaw error; blue dots — where they end; grey rings — before the drive

y — the sideways error, m; θ — the yaw error, rad: ±0.1 rad ≈ ±6° (example); the minus — after the step

F — the step as a table: a row per new value, a column per old one. Not a covariance: the motion

P, P^- — the covariance of (y, θ) before and after the step

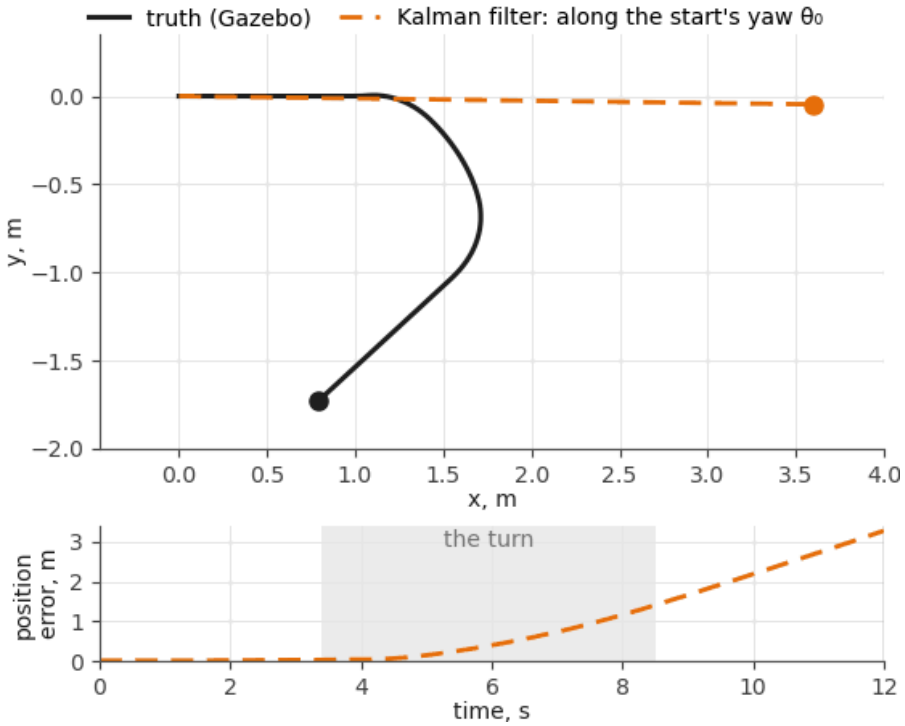
$F^\top$ — F turned over: rows become columns

$FPF^\top$ — the doubt through the step: an error times a has the variance $a^2\sigma^2$, so F stands on both sides

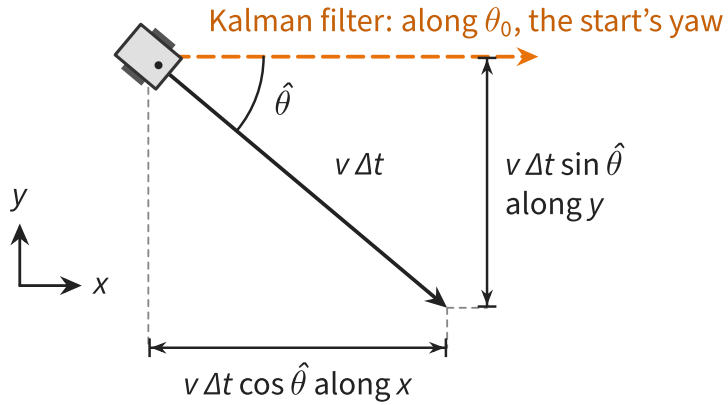
2 m × θ: for small angles, $\sin\theta \approx \theta$

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

THE KALMAN FILTER LOSES A TURNING ROBOT



The drive, by command: 1 m straight, a right U-turn, 1 m back. Known: the speed *v* (wheel odometry) and the yaw (gyroscope). Nobody measures *x*, *y*.



$$x^- = \hat{x} + v\,\Delta t \cos\theta_0 \qquad y^- = \hat{y} + v\,\Delta t \sin\theta_0$$

$$x^- = \hat{x} + v\,\Delta t \cos\hat{\theta} \qquad y^- = \hat{y} + v\,\Delta t \sin\hat{\theta}$$

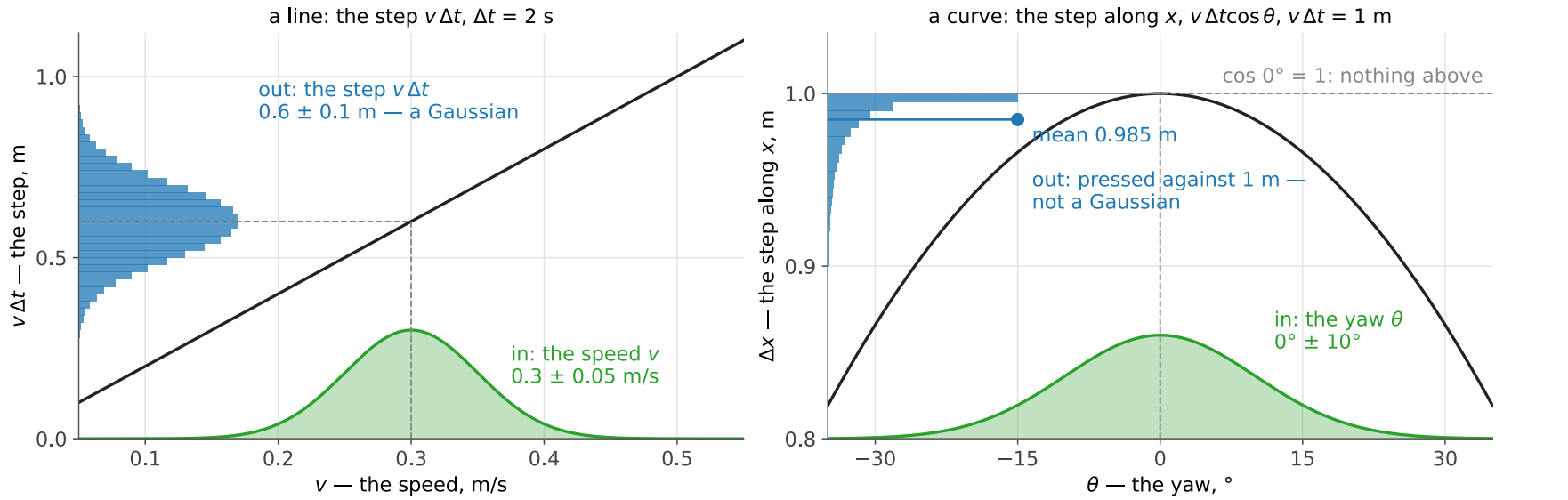
The robot's step turns with $\hat{\theta}$: $\cos\hat{\theta}$, $\sin\hat{\theta}$ are curves. The Kalman filter needs a line, $x^- = F\hat{x} + Bu$ with fixed F, B — the fixed θ_0 gives one.

LEGEND

black — Gazebo's ground truth; **orange dashed** — the Kalman filter stepping along θ_0 (`filterpy`); dots — now position error — the distance from the filter's (x, y) to the truth, m; grey band — the turn
x^-, y^- — the predicted position; $\hat{x}$, $\hat{y}$ — the estimates, m
$v\,\Delta t$ — one step: speed × time
$\hat{\theta}$ — the yaw now; θ_0 — the yaw at the start, a fixed number
F, B — the Kalman filter's fixed tables; u — the step's input, $v\,\Delta t$

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/11-Extended-Kalman-Filters.ipynb

A GAUSSIAN THROUGH A LINE AND THROUGH A CURVE



	A LINE: THE STEP $v\,\Delta t$	A CURVE: THE STEP ALONG x, $v\,\Delta t\cos\theta$
what comes out	a Gaussian again, σ × 2	not a Gaussian: pressed against 1 m, its mean below
the Kalman filter	carries the doubt exactly: σ^2 × 4	cannot carry it as it is

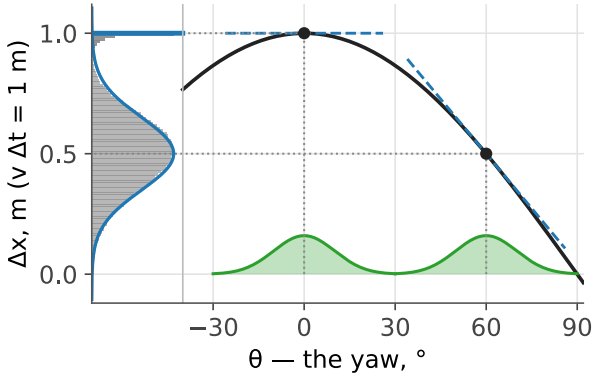
LEGEND

green — what goes in; blue — what comes out: 200 000 random inputs pushed through
black — the line and the curve: what one step does to the number
grey dashed — the input's centre carried through; blue dot — the mean of what comes out
v — the speed; Δt — the step's time; $v\,\Delta t$ — the step, m
$\theta, \hat{\theta}$ — the yaw, its estimate; Δx — the step's part along x
σ — the typical error, the ± on the picture; σ^2 — the doubt, its variance
a line with slope 2: σ × 2, σ^2 × 4

- **Conclusion:** a turning robot steps by $\cos\hat{\theta}$ and $\sin\hat{\theta}$ — curves — so the plain Kalman filter cannot carry its doubt; the EKF fixes this next

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/09-Nonlinear-Filtering.ipynb

THE EKF REPLACES THE CURVE WITH A TANGENT LINE



Kalman filter: $\Delta x = v\,\Delta t \cos\theta_0$ a line: θ_0 is fixed

EKF: $\Delta x = v\,\Delta t \cos\hat{\theta}$ a curve: it moves the estimate

its tangent line at $\hat{\theta}$ carries the doubt:

$$\Delta x \approx v\,\Delta t \cos\hat{\theta} \;-\; v\,\Delta t \sin\hat{\theta}\cdot(\theta-\hat{\theta})$$

A tangent line is straight, so the Kalman filter's rule works again; its slope, blue, goes into F.

LEGEND

black — $\cos\theta$, the curve; **blue dashed** — its tangent lines at 60° and 0°; **green** — the yaw's Gaussian, ±10°; left — what comes out

Δx — the step along x; $v\,\Delta t$ — the step: speed × time

θ — the yaw; $\hat{\theta}$ — its estimate; θ_0 — the yaw at the start

tangent line — the straight line touching the curve at $\hat{\theta}$, not $\tan\theta$; its slope — the derivative, $(\cos\theta)' = -\sin\theta$

`sy`, `cy`, `cp` — sin, cos of yaw, cos of pitch (1 on the floor);

`x_vel`, `delta_sec` — v, Δt; `y_vel`, `z_vel` — 0

`x_coeff`, `dFx_dY` — the slopes of cos and of x in yaw

C++ · robot_localization/src/ekf.cpp — the predict step of the EKF inside ekf_node

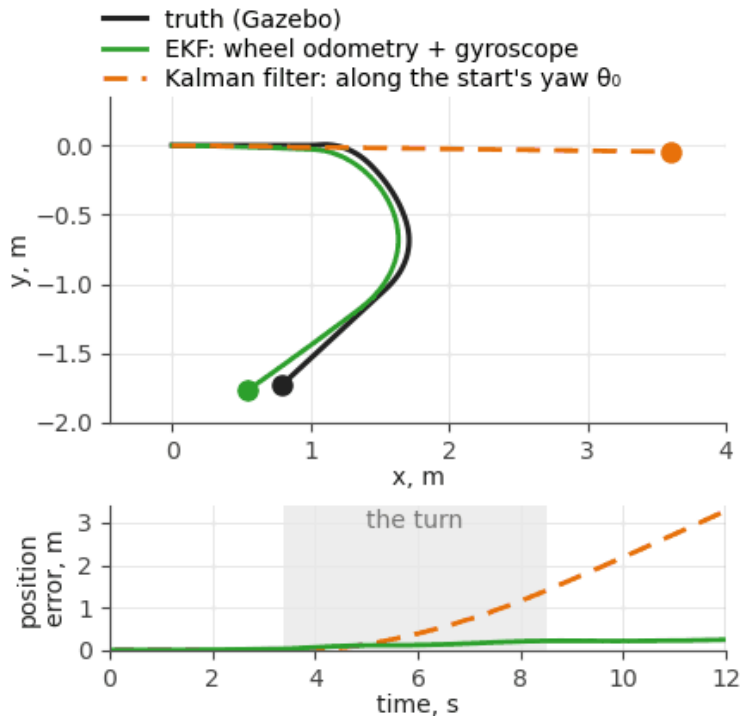
```
// the step itself: x grows by cos(yaw) · vx · Δt — the curve; it moves the estimate
transfer_function_(StateMemberX, StateMemberVx) = cy * cp * delta_sec;
// the tangent line's slope in yaw: -sin(yaw) · v · Δt
x_coeff = -sy * cp;
double dFx_dY =
  (x_coeff * x_vel + y_coeff * y_vel + z_coeff * z_vel) * delta_sec + ...
...
transfer_function_jacobian_(StateMemberX, StateMemberYaw) = dFx_dY;  // into F: row x, column yaw
```

🎯 github.com/cra-ros-pkg/robot_localization

KALMAN FILTER VS EKF: ONLY THE **PREDICT** CHANGES

	KALMAN FILTER	EKF
the state	the distance to the wall, d	the pose (x, y, θ)
predict: the estimate	a line: $d^- = \hat{d} - v\,\Delta t$	a curve: $x^- = \hat{x} + v\,\Delta t \cos\hat{\theta}$
predict: the doubt, $P^- = FPF^\top + Q$	F fixed: 1	F — the tangent lines' slopes at $\hat{\theta}$, new every step
K and the update	the same	the same
implementation	`filterpy` `KalmanFilter`	`robot_localization` `ekf_node`

- **Conclusion:** only predict changes — the estimate moves along the curve, its doubt through the tangent line



The same drive: 1 m, a right U-turn, 1 m back.

LEGEND

black — the truth; orange dashed — the Kalman filter; green — the EKF

position error — the distance from the filter's (x, y) to the truth, m

d — the distance to the wall; x, θ — the position, the yaw

a hat — the estimate; a minus — the prediction, «before the measurement»

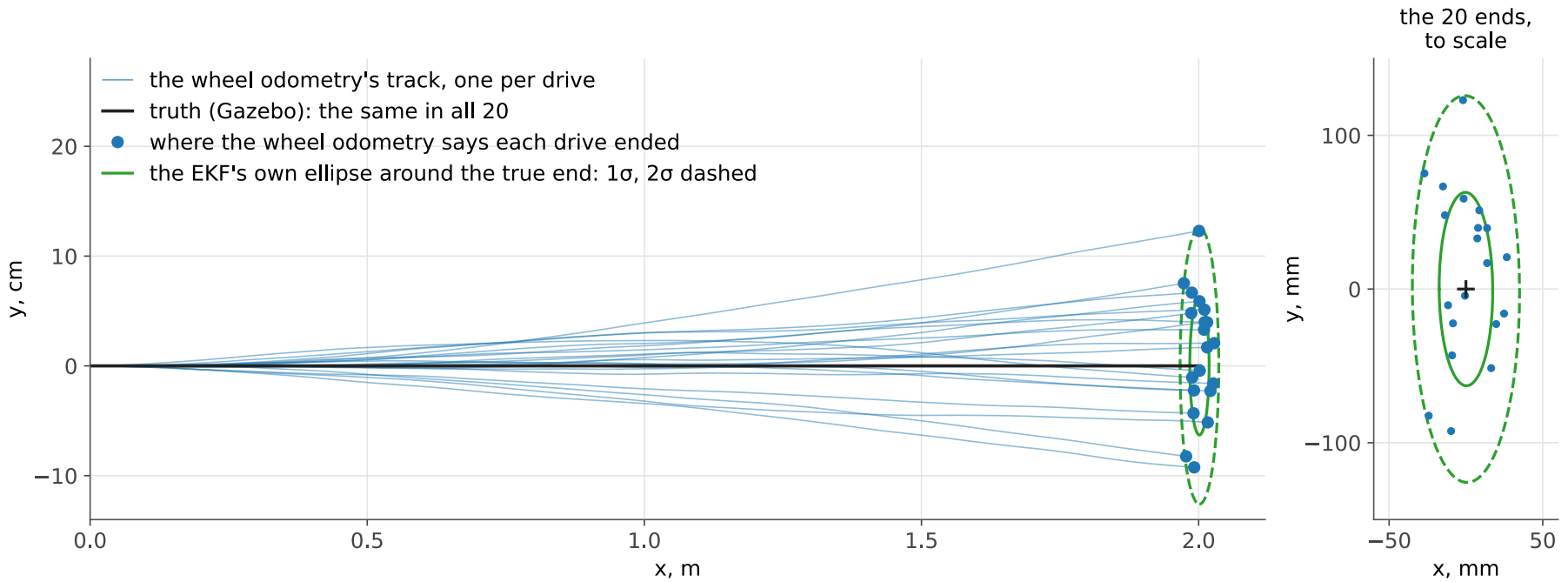
$v\,\Delta t$ — one step

P, P^- — the doubt (covariance) of the estimate, of the prediction

F — the step's slopes; $F^\top$ — F turned over; Q — what a step adds, set by us; K — the Kalman gain

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/11-Extended-Kalman-Filters.ipynb

MONTE CARLO: 20 STRAIGHT DRIVES CHECK THE EKF'S ELLIPSE

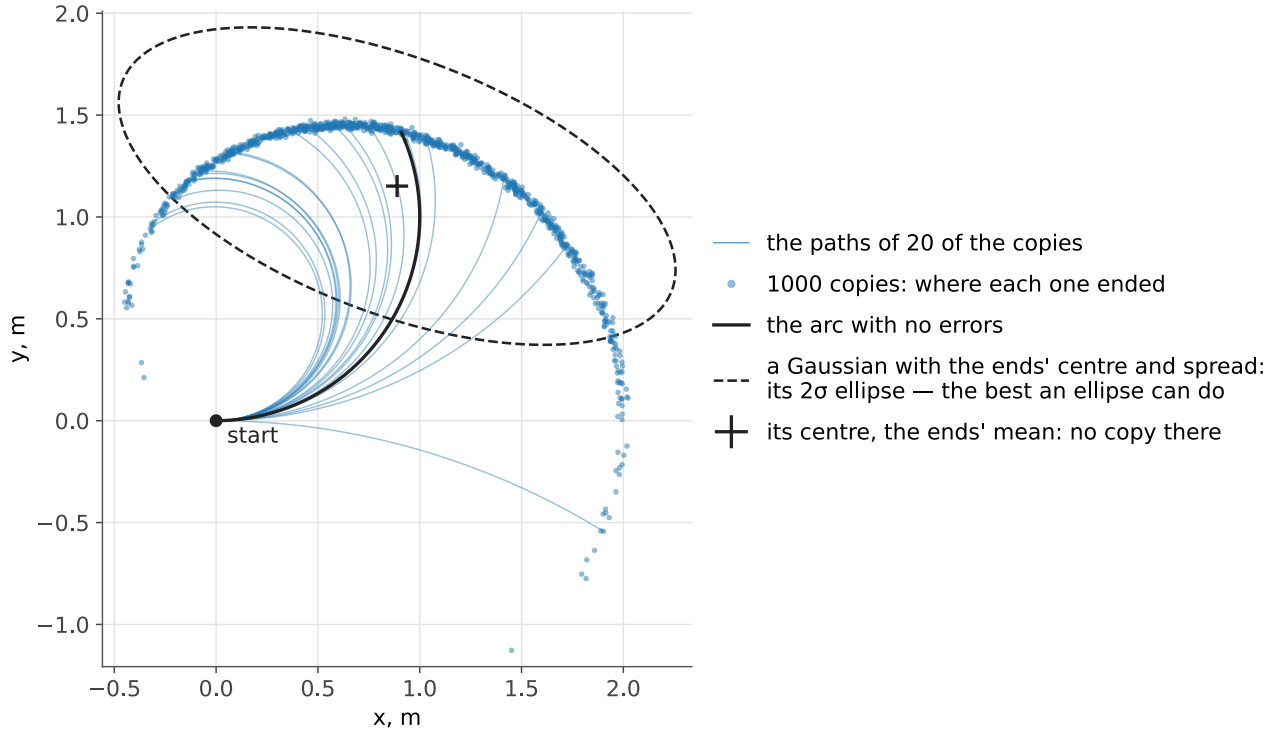


The same 2 m straight drive, 20 times in Gazebo: one truth, only the random noise on the wheel odometry's readings differs. The EKF: `ekf_node` on the wheel odometry alone.

- **Conclusion:** 19 of the 20 ends lie inside the EKF's 2σ ellipse — on a straight drive its doubt is honest

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/09-Nonlinear-Filtering.ipynb

MONTE CARLO ON AN ARC: THE ENDS BEND

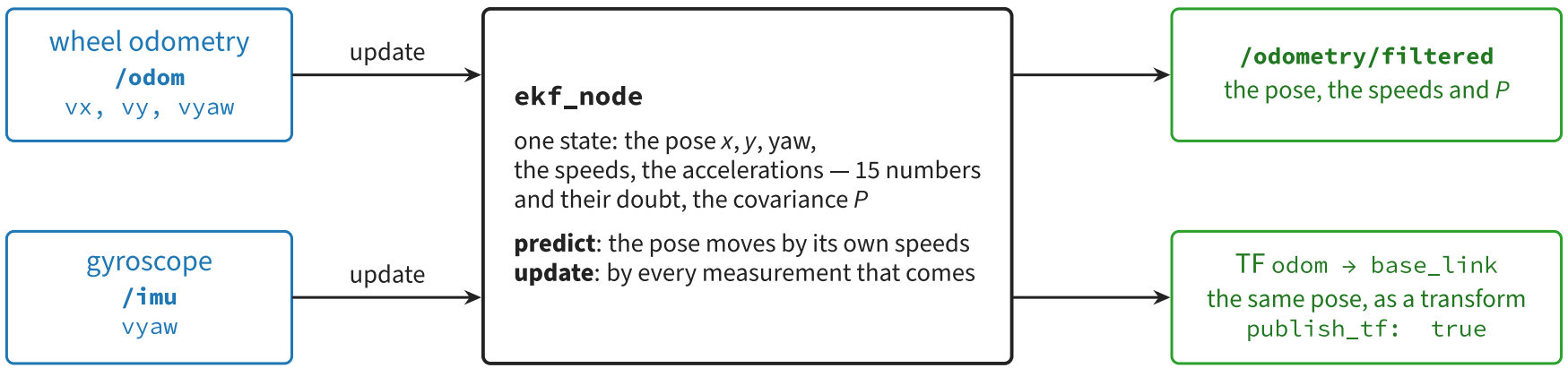


The robot drives a 2 m arc to the left, 1000 times in Python: each copy with its own random errors — of the speed at every step, of the turn rate for the whole drive.

- **Conclusion:** on an arc the ends bend into a thin crescent, and no ellipse fits it — there the EKF's Gaussian is only an approximation

github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/09-Nonlinear-Filtering.ipynb

ROBOT_LOCALIZATION: WHAT GOES IN AND OUT



- **Given:** the speeds from the wheel odometry and the gyroscope · **wanted:** the pose (x, y, θ) and its doubt P
- **Fuse speeds, not poses:** the wheel odometry's pose drifts, its speed is a fresh reading every time; `vy` = 0 is a reading too — a wheeled robot does not drive sideways
- The wheel odometry is a **measurement** here, in update; at the wall it was the predict

LEGEND

blue — what goes in; green — what comes out

`vx`, `vy`, `vyaw` — the speed forward, sideways and of turning

(x, y, θ) — the pose: the position and the yaw

P — the covariance: the doubt of every number of the state

TF `odom → base_link` — where the robot's body is from the place the odometry started

`publish_tf` — whether `ekf_node` sends that TF

docs.nav2.org/jazzy/configuration_and_development/first_time_robot_setup_guide/odom/setup_robot_localization/

github.com/cra-ros-pkg/robot_localization

EKF.YAML: WHICH VARIABLES TO FUSE

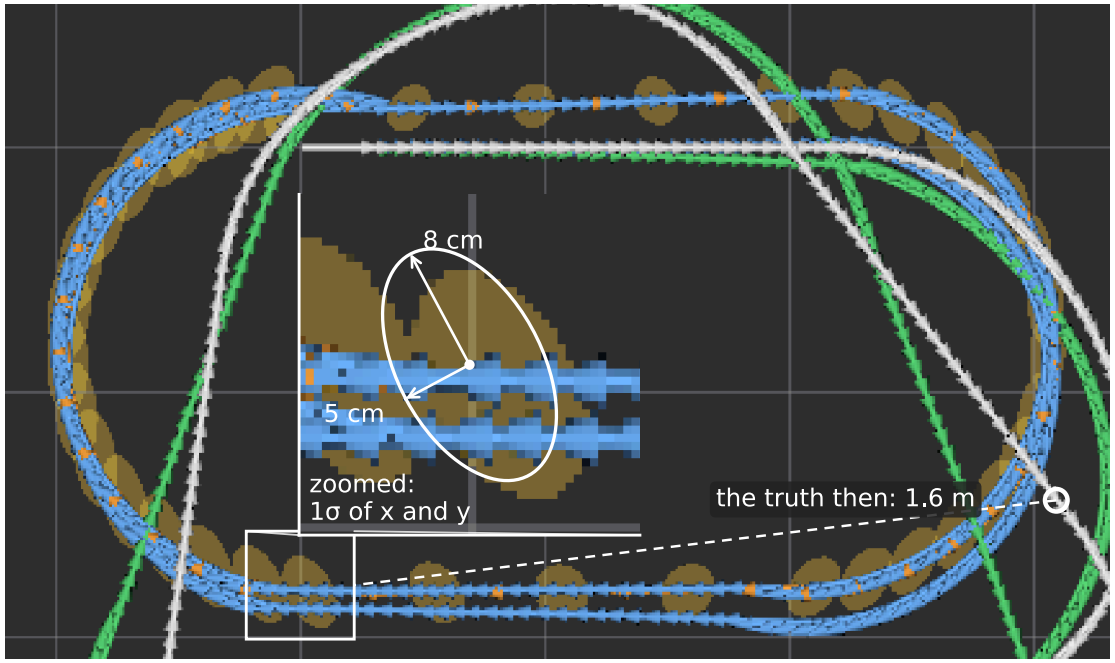
YAML · ekf.yaml — the parameters of robot_localization's ekf_node (excerpt)

```yaml
odom0: /odom
# true — a measurement from /odom, false — ignored
odom0_config: [false, false, false,   # x, y, z
               false, false, false,   # roll, pitch, yaw
               true,  true,  false,   # vx, vy, vz
               false, false, true,    # vroll, vpitch, vyaw
               false, false, false]   # ax, ay, az
imu0: /imu
# the same 15 flags: only vyaw is true
imu0_config:  [false, false, false, false, false, false,
               false, false, false, false, false, true,
               false, false, false]
two_d_mode: true    # a flat floor: z, roll, pitch fused as 0
publish_tf: true    # odom → base_link comes from here
```

- **How many `true`?** Any, all 15 too: each is one more number in update. Only what the sensor really measures — a `true` where the driver sends 0 pulls the estimate to 0
- **One source per absolute value** — `x`, `y`, `yaw`: two that disagree make the estimate jump
- **`two_d_mode: true`** — the filter fuses 0 for `z`, roll, pitch and their speeds: the estimate stays on the floor. Without it nothing measures them, and their doubt grows without bound

▶ docs.nav2.org/jazzy/configuration_and_development/first_time_robot_setup_guide/odom/setup_robot_localization/
🎯 github.com/cra-ros-pkg/robot_localization

ROBOT_LOCALIZATION DRAWS THE ELLIPSE

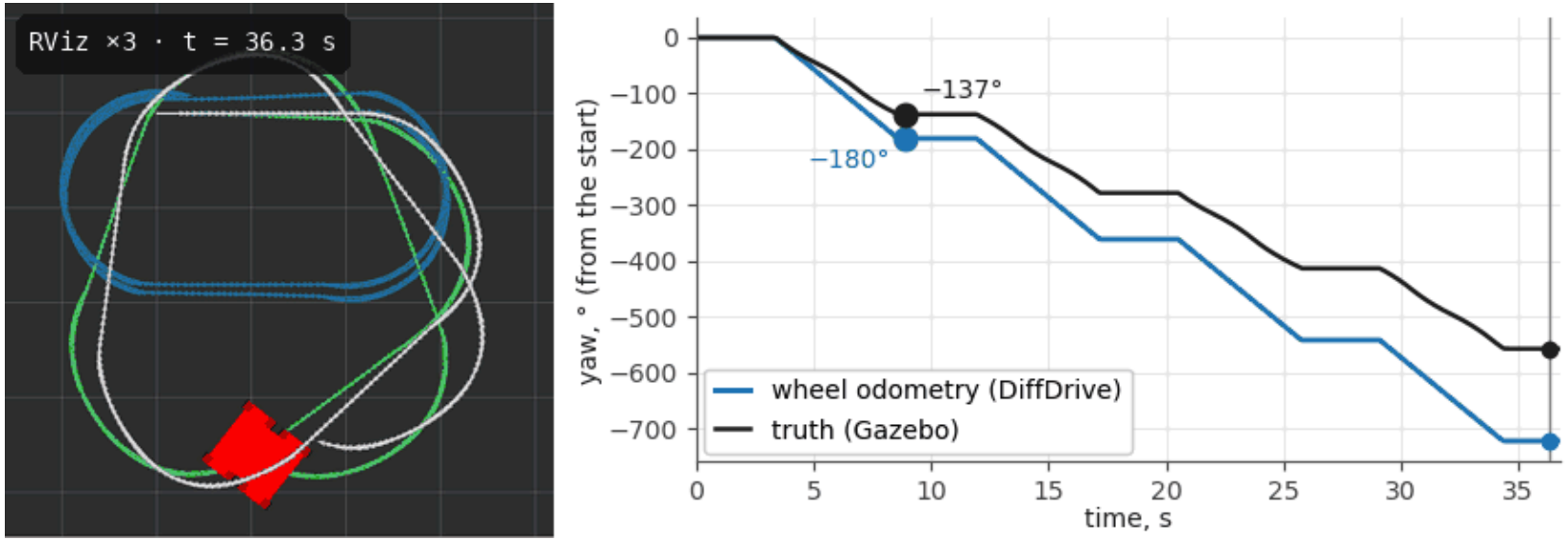


RViz after two loops: white — the truth · blue — the wheel odometry · yellow — the EKF on the wheel odometry alone: its 1σ ellipses · green — the EKF with the gyroscope, later

- **The run:** two loops by command · `ekf_node` on the wheel odometry's speeds alone · **wanted:** the pose and its doubt P
- The ellipses **grow**: nobody measures x, y
- **Conclusion:** the ellipse is small, the truth far away — the filter does not know the wheels slip

docs.nav2.org/jazzy/configuration_and_development/first_time_robot_setup_guide/odom/setup_robot_localization/

FOUR WHEELS SLIP IN TURNS

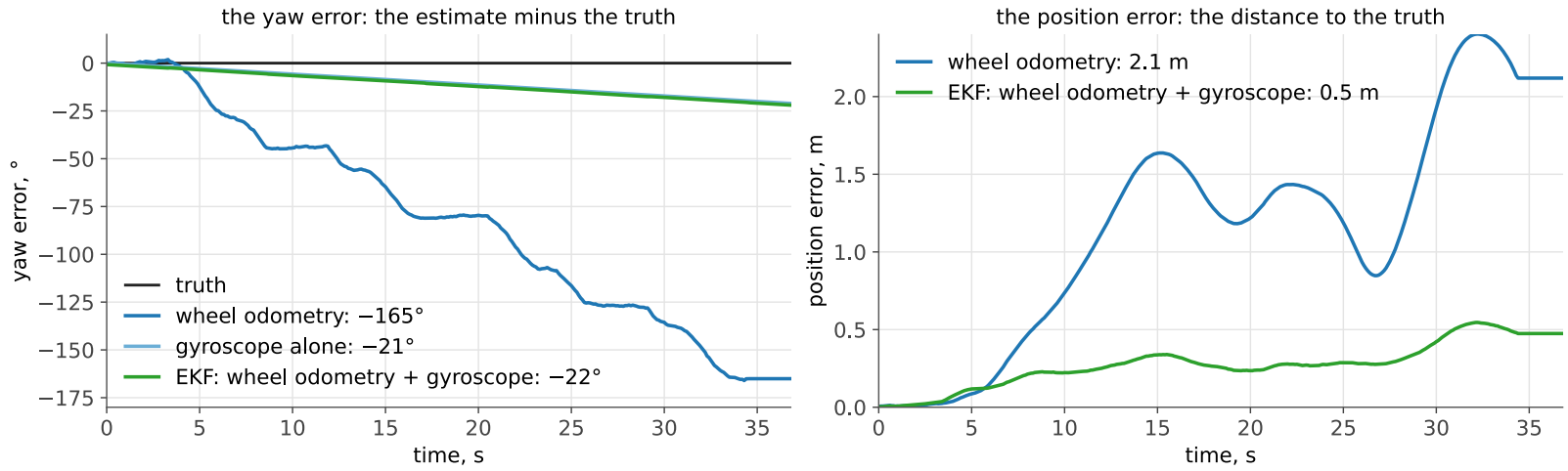


left, RViz ×3: white — the truth, blue — the wheel odometry, green — the EKF with the gyroscope, red — the robot model · right — the yaw: blue — the wheel odometry, black — the truth; the cursor — the frame's time

- Four wheels cannot all roll in a turn: they **slip**, and the wheel odometry counts each turn as if they did not
- **Conclusion:** the wheel odometry's yaw is wrong after every turn, and the EKF on it alone stays sure — its P knows only the noise it is told. We need a sensor of the turn itself: the gyroscope

🎯 github.com/gazebosim/gz-sim/issues/872

WHEEL ODOMETRY + GYROSCOPE: FUSION ON THE LOOP



The same two loops · `ekf_node` fuses the wheel odometry's speeds and the gyroscope's turn rate: in `imu0_config` only `vyaw` is `true`

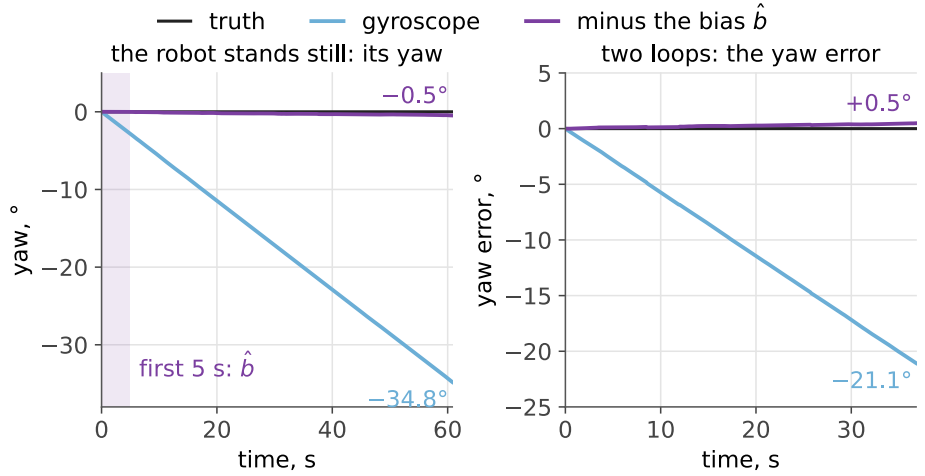
	WHEEL ODOMETRY	GYROSCOPE
how it errs	slips in turns	bias: the same small error in every reading
its yaw error grows with	the number of turns	time — even standing

- **Conclusion:** the EKF takes the yaw from the gyroscope and so fixes the turns; what stays is the gyroscope's bias

MEASURE THE GYROSCOPE'S BIAS WHILE THE ROBOT STANDS

$$\Delta\theta = b\,t \qquad \hat{b} = \text{the mean of } \omega \text{ while the robot stands} \qquad \omega - \hat{b}$$

The yaw error grows in proportion to time, even standing; standing, the true rate is 0, so the gyroscope shows its bias.



C++ · complementary_filter.cpp — imu_tools: the bias estimate inside complementary_filter_node

```
// standing still?
steady_state_ = checkState(ax, ay, az, wx, wy, wz);
if (steady_state_)
{
    ...
    wz_bias_ += bias_alpha_ * (wz - wz_bias_);
}
...
// every step: the rate minus the bias
double wz_unb = wz - wz_bias_;
```

- `imu_tools` does it by itself: `do_bias_estimation: true`; its `/imu/data` carries $\omega - \hat{b}$ — give that to `robot_localization`

LEGEND

light blue — the gyroscope's yaw; purple — with $\hat{b}$ subtracted; purple strip — the first 5 s, where $\hat{b}$ is measured; black — the truth

$\Delta\theta$ — the yaw error; t — time

b — the bias: the same small error in every reading; $\hat{b}$ — the bias, measured

ω — the gyroscope's turn rate, rad/s; $\omega - \hat{b}$ — without the bias

`checkState` — does the robot stand? `ax` … `wz` — the IMU's readings

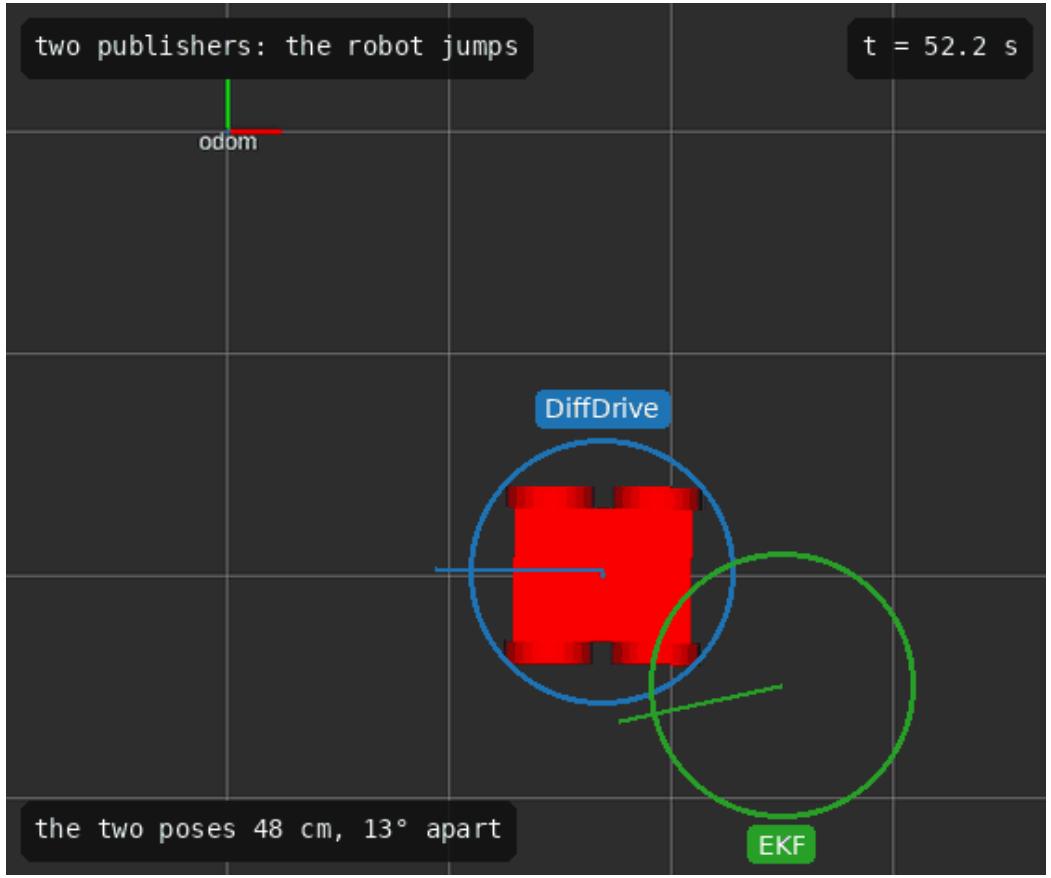
`wz_bias_` — $\hat{b}$, a running mean; `bias_alpha_` — a new reading's weight, 0.01

`wz_unb` — $\omega - \hat{b}$

🎯 github.com/CCNYRoboticsLab/imu_tools

THE EKF RUNS, AND THE ROBOT IN RVIZ JUMPS BETWEEN TWO PLACES. WHY?

ANSWER: TWO NODES PUBLISH ONE TF LINK




```
$ ros2 topic info /tf --verbose     # who publishes on /tf
Publisher count: 3
Node name: robot_state_publisher
...
Node name: ekf_filter_node
...
Node name: ros_gz_bridge
...
```

YAML · bridge.yaml — ros_gz_bridge's topics: DiffDrive's TF taken out

```
# - ros_topic_name: "/tf"
#   gz_topic_name: "/model/demo_bot/tf"
#   ros_type_name: "tf2_msgs/msg/TFMessage"
#   gz_type_name: "gz.msgs.Pose_V"
#   direction: GZ_TO_ROS
```

```
$ ros2 topic info /tf --verbose
Publisher count: 2                  # robot_state_publisher, ekf_filter_node
```

- `/tf` may have many publishers: each sends its own links. **One TF link — one publisher**

docs.nav2.org/jazzy/configuration_and_development/first_time_robot_setup_guide/odom/setup_robot_localization/

THE FILTER FAMILY: BELIEF, PREDICT, UPDATE

FILTER	THE BELIEF	PREDICT	UPDATE	IMPLEMENTATION
complementary	one angle	+ the gyroscope's turn	a fixed share of the gap to the accelerometer	`imu_complementary_filter`
Bayes on a grid	any shape, on cells	move and blur every cell	× the likelihood, rescale	`filterpy.discrete_bayes`
Kalman	one Gaussian	along a straight line	the weight K	`filterpy` `KalmanFilter`
EKF	one Gaussian	along a curve; the doubt through the tangent line	the weight K	`robot_localization` `ekf_node`
UKF (unscented Kalman filter), not covered	one Gaussian	along a curve; the doubt through a few chosen points	the weight K	`robot_localization` `ukf_node`
particles	a cloud of guesses	move each guess, with noise	weigh each by the likelihood, keep the likely	later: SLAM

- **Conclusion:** a few numbers, a centre and P — Kalman, EKF, UKF; many, of any shape — the grid, particles
- **Tracking:** we know where we started. **"Where am I on the map?"** needs many guesses — the next lecture

📖 porabook.com/book/

Lecture 8 — SLAM

A map from scans · where am I on that map · both at once — the simplest SLAM

📖 porabook.com/book/