`sftools` Manual

TSRT14 Sensor Fusion

November 13, 2026

Contents

1 Overview **2**
- 1.1 Shapes and conventions . . . 3
- 1.2 Suggested learning path . . . 3
- 1.3 Good practice checklist . . . 3

2 State containers **4**

3 Dynamic models **6**

4 Observation models **7**

5 Estimators **8**
- 5.1 Linear least squares . . . 9
- 5.2 Nonlinear least squares . . . 10
- 5.3 Kalman filtering . . . 12
- 5.4 Good practice example: Innovation monitoring and outlier rejection . . . 14
- 5.5 The same loop with UKF and particle filter . . . 17

6 Moment transforms and analysis helpers **19**
- 6.1 Analysis helpers: CRLB map . . . 21

7 Working on exercises **22**
- 7.1 Participant starter template (recommended workflow) . . . 22
- 7.2 Common pitfalls (and fast fixes) . . . 23
- 7.3 API quick reference . . . 24

About this document

The examples are run when this manual is built, so the displayed output and figures come directly from the accompanying code. In PDF viewers that support it, copying code preserves the source text and indentation rather than visual line wraps. Selecting part of a line may copy the whole source line; check line breaks when copying across pages.

1 Overview

`sftools` (distributed as `sensorfusion-tools`) is a Python toolbox for learning estimation, filtering, and sensor fusion. It is used in the TSRT14 Sensor Fusion course at Linköping University (LiU) and is developed by Gustaf Hendeby, who has taught the course for more than ten years. The toolbox grew out of that teaching experience and a wish to make sensor fusion more accessible.

The teaching focus shapes the toolbox's design. Rather than hide an estimation problem behind a single high-level call, the toolbox makes its structure visible through explicit workflows. Each step remains in your hands and in plain sight: for example, you write the filtering loop yourself, calling measurement and time updates separately.

To support these workflows, the toolbox builds on NumPy and SciPy for numerical work and provides four kinds of building blocks:

1. **State containers** — `GaussianState`, `ParticleState`, `StateTrajectory`.
2. **Dynamic models** (`sftools.model_dyn`) — state-transition models (`linear`, `cv`, `ct`), created via `create_dynamics(name, ...)`.
3. **Observation models** — measurement models (`linear`, `nonlinear`, `pos`, `toa`, `tdoa`, `tdoa_r0`), created via `create_observation(name, ...)`.
4. **Estimators and transforms** (`sftools.estimator`, `sftools.helper`) — `KalmanFilter`, `UnscentedKalmanFilter`, `ParticleFilter`, least-squares solvers (`ls`, `wls`, `nls`), and moment transforms (`ut`, `tt`, `mct`).

These building blocks can be combined to simulate measurements, estimate states, and propagate uncertainty. To inspect the results, `sftools.plot` uses Matplotlib to display point estimates, signals, and trajectories, including confidence ellipses and bands.

This manual follows the same workflow through small examples you can adapt while solving the exercises, starting with the shapes and conventions shared by the interfaces. Note that it assumes familiarity with the sensor-fusion concepts covered in the course; it is not a standalone textbook. For full signatures and advanced options, use Python's `help()`, for example `help(sftools.create_observation)`.

Getting the toolbox

The toolbox is available from PyPI. It requires Python **3.14 or later**. In an existing uv project, install it with `uv add sensorfusion-tools`. The package name is `sensorfusion-tools`, but the

Python import is `import sftools`. For project setup and a first example, see the getting-started guide.

1.1 Shapes and conventions

Model calculations represent states, observations, and sample collections as columns. Most interface mistakes are shape mistakes, so keep this table at hand: Here, `n` is the state dimension, `m` the measurement dimension, and `N` the number of samples or time steps.

Quantity	Shape
One state	`(n, 1)`
One observation	`(m, 1)`
State or sample collection	`(n, N)`
Predicted observations for a collection	`(m, N)`
Measurement Jacobian at one state	`(m, n)`
Measurement covariance	`(m, m)`

`ls`, `wls`, `nls`, `helper.llh`, and the filter measurement updates require a single observation in column form. They reject flat, row, and batched observations rather than silently reshaping them — convert a flat observation explicitly with `y[:, None]`.

`GaussianState.x` stores its mean as a flat `(n,)` array; `state.mean()` returns it as an `(n, 1)` column. Built-in models also accept flat state inputs, while custom model callbacks must return the shapes above.

1.2 Suggested learning path

If this is your first time using `sftools`, follow this order:

1. **State containers**: understand shapes and conventions.
2. **Dynamic/observation models**: verify `f/h` and `F/H` behavior on simple inputs.
3. **Least-squares solvers**: build intuition for static estimation.
4. **Recursive filtering**: combine models and estimators over time.
5. **Diagnostics and plotting**: validate if your estimator behaves as expected.

The examples in this manual are deliberately small and reproducible, so that you can adapt them quickly in your own exercise scripts.

1.3 Good practice checklist

Use this checklist each time you implement a new exercise:

- Keep your work in a saved Python script (`.py`) or Jupyter notebook (`.ipynb`), rather than only in an interactive console. Include the setup, calculations, and plots, with brief notes on your choices, so you can rerun the work and return to it later. Check that it runs from start to finish in a fresh session.
- Fix random seeds (`np.random.default_rng(seed)`) for reproducible results.
- Keep state vectors as explicit column vectors `(n, 1)`.
- Print matrix shapes early (`x.shape`, `P.shape`, `H.shape`) to catch interface mismatches.
- Start with conservative covariance (`P0`) rather than overconfident priors.
- Track a metric (for example RMSE or innovation norm) while tuning.
- Visualize both estimate and uncertainty; a trajectory alone can hide filter inconsistency.

Code: Common imports

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

import sftools
from sftools import create_dynamics, create_observation, estimator, helper, plot
from sftools.types import GaussianState, ParticleState, StateTrajectory

print(f"sftools version: {sftools.__version__}")
```

Output: Common imports

```
sftools version: 2026.0.3
```

2 State containers

`sftools.types` provides three containers used by every estimator and plotting helper in the library:

- `GaussianState(x, P)` — a mean/covariance pair, for Kalman-family filters. `.mean()` returns an `(n, 1)` column vector, `.cov()` returns `(n, n)`.
- `ParticleState(x, logw)` — weighted samples (`x`: `(n, N)`, `logw`: `(N,)`), for particle filters. `.mean()`/`.cov()` compute the weighted empirical moments.
- `StateTrajectory` — a time series of means (and optionally covariances), used to record filter output or ground truth for plotting.

Code: Gaussian and particle states

```
x0 = np.array([1.0, 2.0])
P0 = np.diag([0.5, 0.2])
state = GaussianState(x0, P0)
print("Gaussian mean:", state.mean().ravel())
print("Gaussian cov:\n", state.cov())

rng = np.random.default_rng(0)
particles = rng.multivariate_normal(mean=x0, cov=P0, size=500).T
logw = np.full(500, -np.log(500))
pstate = ParticleState(particles, logw)
print("\nParticle mean:", pstate.mean().ravel())
print("Particle cov:\n", np.round(pstate.cov(), 3))
```

Output: Gaussian and particle states

```
Gaussian mean: [1. 2.]
Gaussian cov:
 [[0.5 0. ]
 [0.  0.2]]

Particle mean: [0.93790719 1.99631314]
Particle cov:
 [[0.484 0.002]
 [0.002 0.187]]
```

Code: State trajectories

```
traj = StateTrajectory.preallocate(state_dim=2, steps=5, with_cov=True)
for k in range(5):
    traj.write_state(k, GaussianState(x0 + k, P0))
print("trajectory.mean shape:", traj.mean.shape)
print("component 0 over time:", traj.component(0))
```

Output: State trajectories

```
trajectory.mean shape: (2, 5)
component 0 over time: [1. 2. 3. 4. 5.]
```

Iterating over a `StateTrajectory` yields the mean states as `(n, 1)` columns, ready to pass to a filter's `meas_update`. Use `preallocate_like(other, ...)` to create a trajectory with the same step count (and time stamps) as an existing one, which keeps measurement and estimate trajectories aligned. See `help(StateTrajectory)` for the full interface, including `iter_samples` and the `from_*` constructors.

Try it yourself: State containers

1. Change `P0` so one axis has much larger uncertainty and inspect `ParticleState.cov()`.
2. Increase the number of particles from 500 to 5000 and compare how stable the empirical mean/covariance become.
3. Preallocate `StateTrajectory` with more steps and use `component(1)` to inspect the second state component over time.

3 Dynamic models

Dynamic models are created with `create_dynamics(name, ...)` and all expose the same interface: `f(x)` (deterministic propagation), `F(x)` (Jacobian), `G(x)` (process-noise gain), and `random(n=1, x=None, rng=None)` (stochastic propagation).

Name	State layout	Constructor args
`linear`	user-defined	`F, G, Q`
`cv`	`[position..., velocity...]` (2n states)	`T, Q, n`
`ct`	`[x, y, v, heading, turn-rate]` (5 states)	`T, Q, n` (n unused)

`T` is the sampling interval. `Q` is the covariance of the process noise, whose effect on the state is set by `G(x)`; the added state covariance is `G(x) @ Q @ G(x).T`. In the 2D CV example, `Q` is `(2, 2)` although the state has four components.

Code: Constant-velocity model

```
dyn = create_dynamics("cv", T=0.1, Q=0.01 * np.eye(2), n=2)
x0_cv = np.array([[0.0], [0.0], [1.0], [0.5]])
print("f(x0)  =", dyn.f(x0_cv).ravel())
print("F(x0)  =\n", dyn.F(x0_cv))

rng = np.random.default_rng(1)
samples = dyn.random(n=3, x=x0_cv, rng=rng)
print("\n3 stochastic samples:\n", np.round(samples, 3))
```

Output: Constant-velocity model

```
f(x0)  = [0.1  0.05 1.   0.5 ]
F(x0)  =
 [[1.  0.  0.1 0. ]
 [0.  1.  0.  0.1]
 [0.  0.  1.  0. ]
 [0.  0.  0.  1. ]]
```

```
3 stochastic samples:
 [[0.1   0.1   0.1  ]
 [0.05  0.049 0.05 ]
 [1.003 1.003 1.009]
 [0.508 0.487 0.504]]
```

Code: Coordinated-turn model

```
ct = create_dynamics("ct", T=0.5, Q=np.diag([0.01, 0.001]), n=0)
xc0 = np.array([[0.0], [0.0], [1.0], [0.0], [0.3]])
print("CT f(x0) =", ct.f(xc0).ravel())
```

Output: Coordinated-turn model

```
CT f(x0) = [0.49812711 0.03742974 1.         0.15       0.3       ]
```

Try it yourself: Dynamic models

1. Change `T` from `0.1` to `0.5` in the CV model and inspect the new transition matrix `F(x0)`.
2. Increase/decrease `Q` and compare how spread out `dyn.random(...)` samples become.
3. For the CT model, vary turn-rate from `0.0` to `0.5` and observe how `f(x0)` changes.

4 Observation models

Observation models are created with `create_observation(name, ...)` and expose `h(x)` (expected measurement, batchable over `(n, N)` states), `H(x)` (Jacobian at a single state), and `random(x, n=1, rng=None)`. The measurement-noise covariance is available as `model.R`; diagonal entries are variances, not standard deviations.

Name	Use case	Key args
`linear`	`y = Hx + v`	`H, R`
`nonlinear`	user-supplied `h`/`H` callables	`h, H, R, **kwargs`
`pos`	direct position observation	`R, n`
`toa`	range to each anchor	`p, R, n`
`tdoa`	range differences vs. first anchor	`p, R, n`
`tdoa_r0`	absolute range + unknown offset `r0`	`p, R, n`

Anchor positions are columns of `p`. With `n` anchors, `toa` returns `n` ranges while `tdoa` returns `n-1` differences, so their `R` matrices have shapes `(n, n)` and `(n-1, n-1)`, respectively. For custom nonlinear models, `h` must accept `(n, N)` states and return `(m, N)` measurements; `H` accepts one

state and returns `(m, n)`. Range Jacobians are undefined at an anchor: choose initial guesses and linearization points away from anchor positions.

Code: Time-of-arrival (range) observation model

```
anchors = np.array([[0.0, 10.0, 10.0, 0.0], [0.0, 0.0, 10.0, 10.0]])
obs = create_observation("toa", p=anchors, R=0.01 * np.eye(4), n=4)

x_true = np.array([[3.0], [4.0]])
print("h(x) =", obs.h(x_true).ravel())
print("H(x) =\n", np.round(obs.H(x_true), 3))

rng = np.random.default_rng(2)
y = obs.random(x=x_true, rng=rng)
print("\nnoisy measurement:", y.ravel())

pos_obs = create_observation("pos", R=0.1 * np.eye(2), n=2)
print("\npos model h(x):", pos_obs.h(x_true).ravel())
```

Output: Time-of-arrival (range) observation model

```
h(x) = [5.         8.06225775 9.21954446 6.70820393]
H(x) =
 [[ 0.6    0.8  ]
 [-0.868  0.496]
 [-0.759 -0.651]
 [ 0.447 -0.894]]

noisy measurement: [5.01890534 8.0099829  9.1782381  6.46405719]

pos model h(x): [3. 4.]
```

Try it yourself: Observation models

1. Move one anchor and inspect how `toa` measurements and Jacobian entries change.
2. Increase measurement noise `R` and verify that `random(...)` output becomes visibly noisier.
3. Replace `toa` with `tdoa` in your own script and compare observability in symmetric vs. asymmetric anchor layouts.

5 Estimators

The one-shot least-squares solvers `ls` and `wls` return a `GaussianState`; `nls` does so by default but returns an `NLSResult` with `return_result=True`. The recursive filters are `KalmanFilter` (an EKF), `UnscentedKalmanFilter` (sigma-point propagation via `helper.ut`), and the bootstrap `ParticleFilter`.

5.1 Linear least squares

Use `ls` for unweighted linear least squares and `wls` to account for the measurement covariance `R`. Here, three measurements constrain two state components, and the noise is unequal and correlated.

Code: LS and WLS with a linear observation model

```
H = np.array([[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]])
R = np.array([[0.04, 0.02, 0.0], [0.02, 0.09, 0.0], [0.0, 0.0, 0.01]])
linear_obs = create_observation("linear", H=H, R=R)

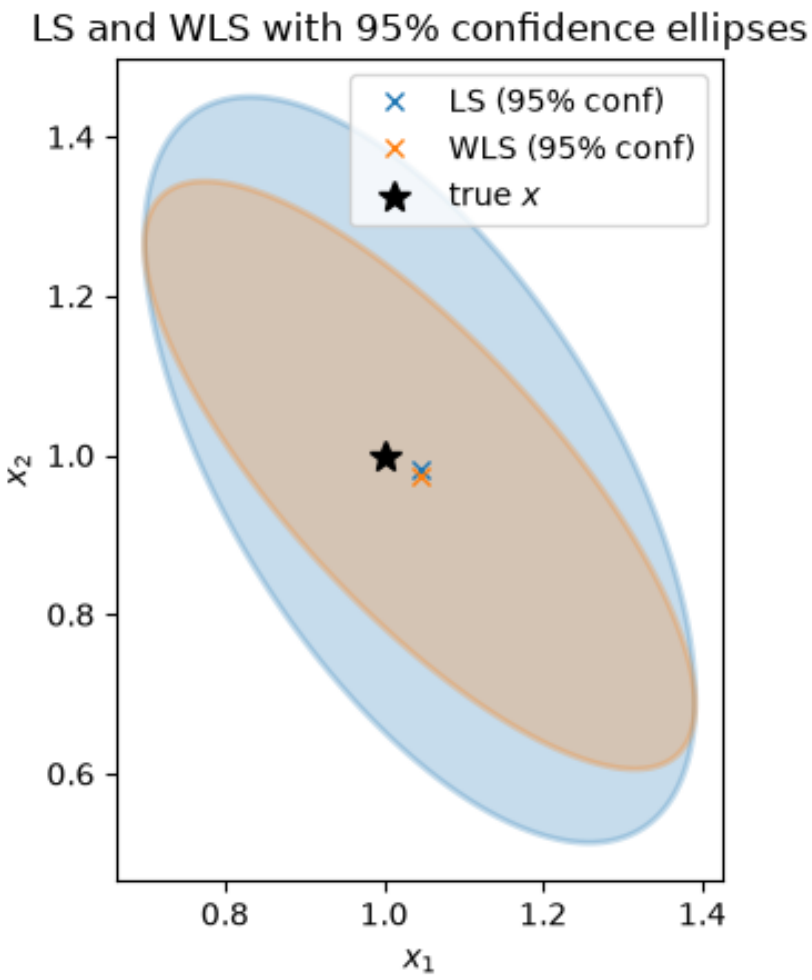
rng = np.random.default_rng(39)
x_true = np.ones((2, 1))
y = linear_obs.random(x=x_true, rng=rng)

xhat_ls = estimator.ls(linear_obs, y)
xhat_wls = estimator.wls(linear_obs, y)
print("LS  estimate:", xhat_ls.mean().ravel())
print("WLS estimate:", xhat_wls.mean().ravel())

fig, ax = plt.subplots()
plot.estimate(xhat_ls, conf=0.95, label="LS", ax=ax)
plot.estimate(xhat_wls, conf=0.95, label="WLS", ax=ax)
ax.plot(x_true[0, 0], x_true[1, 0], "k*", markersize=10, label="true $x$")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
ax.set_title("LS and WLS with 95% confidence ellipses")
ax.set_aspect("equal")
ax.legend()
plt.show()
```

Output: LS and WLS with a linear observation model

```
LS  estimate: [1.04492661 0.98157631]
WLS estimate: [1.04492661 0.97473961]
```



The estimates and ellipses differ because WLS uses `R` to weight the measurements. LS also reports covariance using `R`, but does not use it to weight the estimate. If `R` is a scaled identity, the two methods agree. Both solve a linear problem with design matrix `model.H(x0)`; the optional `x0` is only passed to that Jacobian, not used for a nonlinear correction. Use `nls` for a nonlinear observation model.

5.2 Nonlinear least squares

Reuse the four-anchor geometry from the observation-model example. `nls` evaluates both `h` and `H`, starting from a supplied initial guess.

Code: NLS range localization

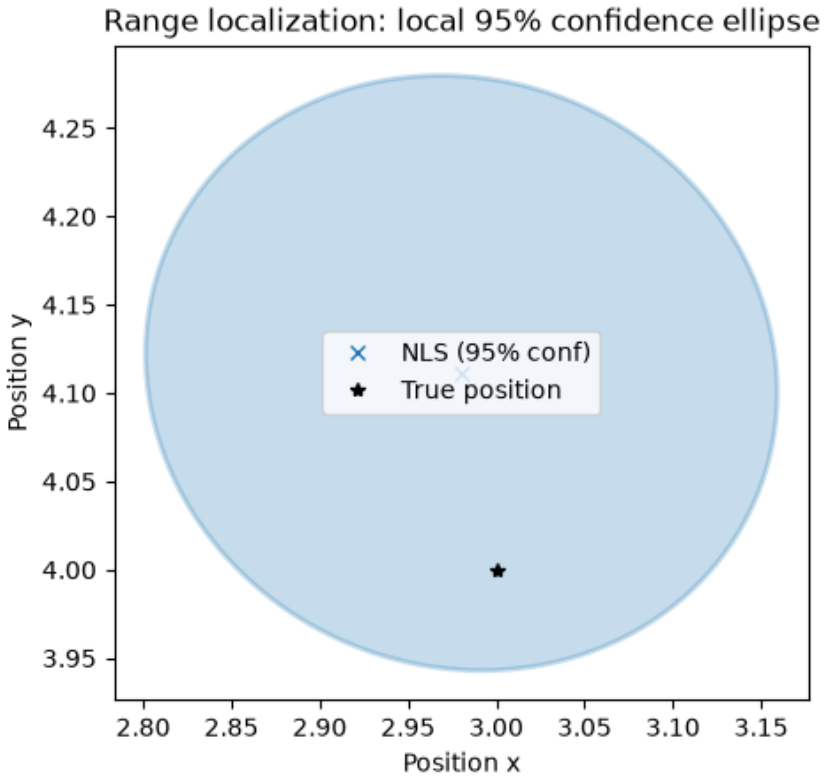
```
range_obs = create_observation("toa", p=anchors, R=0.01 * np.eye(4), n=4)
x_true = np.array([[3.0], [4.0]])
```

```
y = range_obs.random(x=x_true, rng=np.random.default_rng(2))
x_init = np.array([[5.0], [5.0]])
xhat_nls = estimator.nls(range_obs, y, x_init)
print("NLS estimate:", xhat_nls.mean().ravel())

fig, ax = plt.subplots()
plot.estimate(xhat_nls, conf=0.95, label="NLS", ax=ax)
ax.plot(x_true[0, 0], x_true[1, 0], "k*", label="True position")
ax.set_xlabel("Position x")
ax.set_ylabel("Position y")
ax.set_aspect("equal")
ax.set_title("Range localization: local 95% confidence ellipse")
ax.legend()
plt.show()
```

Output: NLS range localization

```
NLS estimate: [2.98026844 4.11139668]
```



The plot zooms in on the estimate; the anchors are outside the view. The ellipse describes local

covariance, not a guarantee of global optimality or confidence-region coverage. Try different initial guesses when the geometry permits multiple solutions.

By default, `nls` returns a `GaussianState` and raises on solver failure or singular local covariance. Set `return_result=True` to request an `NLSResult` and inspect SciPy's optimizer diagnostics. In that mode, `estimate` is `None` when no estimate can be formed.

Try it yourself: Static localization with NLS

Use the nonlinear least-squares section as a template and:

1. Simulate multiple measurements from different anchor geometries.
2. Run `nls` from two different initial guesses.
3. Compare convergence and final covariance; discuss why geometry and initial guess matter.

5.3 Kalman filtering

All recursive filters expose exactly two methods: `meas_update(state, obs_model, y)` and `time_update(state)`. There is deliberately no single `filter(...)` call that runs the whole recursion — you write the loop yourself and call the two updates explicitly. This has two purposes: it keeps the two-stroke structure of every recursive filter visible (correct with the observation, then predict to the next step), and it gives you control over each observation, so outliers can be inspected and skipped with ordinary Python (see the gating example below).

A `KalmanFilter` alternates `meas_update` (correction) and `time_update` (prediction). The example below tracks a constant-velocity target from noisy position measurements and records both the ground truth and the filter estimate in `StateTrajectory` objects for plotting.

Code: Kalman filter tracking example

```
rng = np.random.default_rng(0)
dyn = create_dynamics("cv", T=0.1, Q=0.1 * np.eye(2), n=2)
obs = create_observation("pos", R=0.5 * np.eye(2), n=2)

N = 60
x0 = np.array([0.0, 0.0, 1.0, 0.5])
truth = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=False)
meas = StateTrajectory.preallocate(state_dim=2, steps=N, with_cov=False)

x = x0
for k in range(N):
    truth.write_mean(k, x)
    meas.write_mean(k, obs.random(x=x, rng=rng).ravel())
    x = dyn.random(x=x, rng=rng).ravel()

filt = estimator.KalmanFilter(dyn_model=dyn)
state = GaussianState(x=np.zeros(4), P=10 * np.eye(4))
est = StateTrajectory.preallocate_like(meas, state_dim=4, with_cov=True)
```

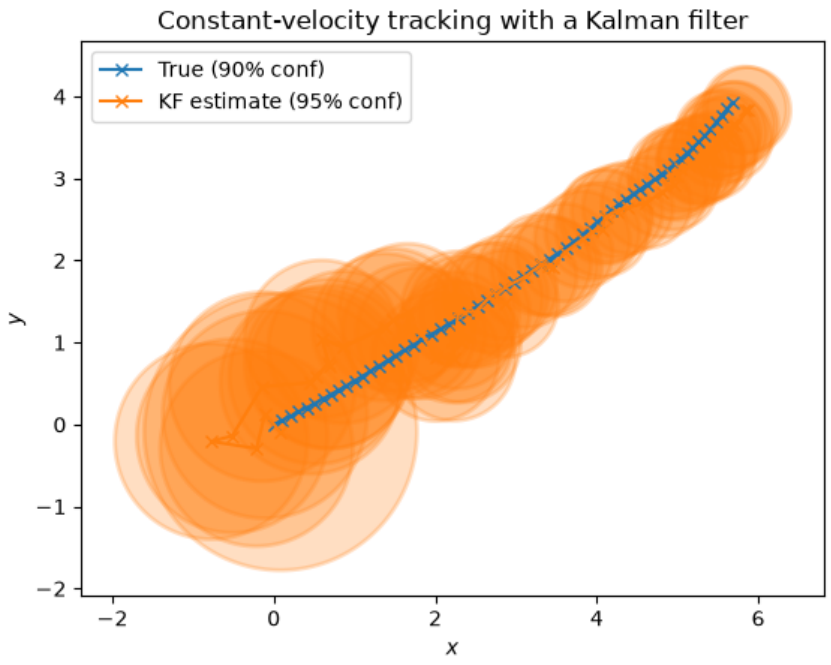
```
for k, y in enumerate(meas):
    state = filt.meas_update(state, obs, y)
    est.write_state(k, state)
    state = filt.time_update(state)

rmse = np.sqrt(np.mean((est.mean[:2] - truth.mean[:2]) ** 2))
print(f"Position RMSE: {rmse:.3f}")

fig, ax = plt.subplots()
plot.trajectory(truth, select=(0, 1), label="True", ax=ax)
plot.trajectory(est, select=(0, 1), conf=0.95, label="KF estimate", ax=ax)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("Constant-velocity tracking with a Kalman filter")
ax.legend()
plt.show()
```

Output: Kalman filter tracking example

```
Position RMSE: 0.265
```



Try it yourself: Filter tuning

Keep the same synthetic trajectory and tune:

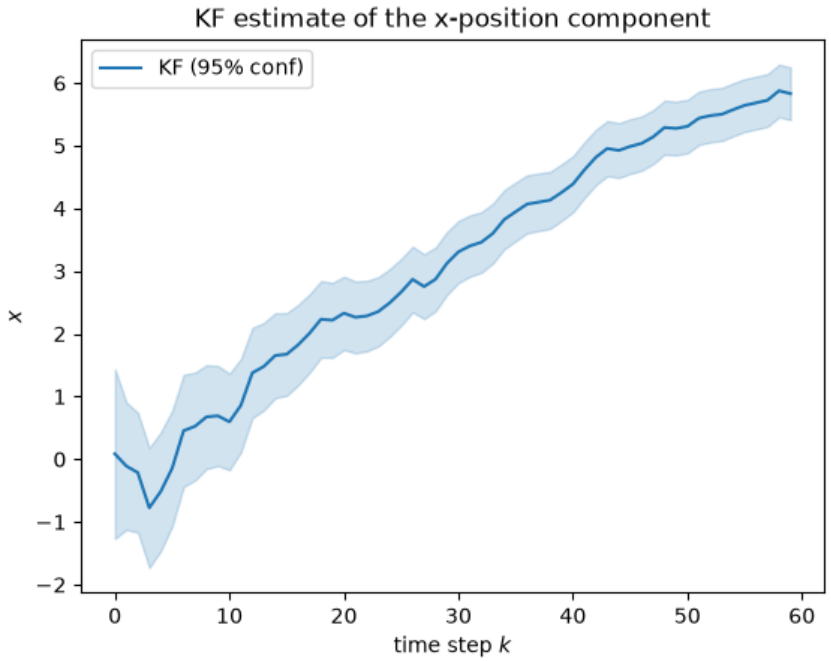
1. Under-modeled process noise (`Q` too small).
2. Over-modeled process noise (`Q` too large).
3. A balanced setting.

For each case, report RMSE and visually inspect confidence bounds. A good setting should give low error without overconfident uncertainty.

The x-position track over time, with its 95% confidence band, can also be plotted directly with `plot.signal`:

Code: Signal plot of the x-position track

```
fig, ax = plt.subplots()
plot.signal(est, select=0, conf=0.95, ax=ax, label="KF")
ax.set_xlabel("time step $k$")
ax.set_ylabel("$x$")
ax.set_title("KF estimate of the x-position component")
ax.legend()
plt.show()
```



5.4 Good practice example: Innovation monitoring and outlier rejection

Real measurements occasionally contain gross errors (multipath, packet loss, synchronization glitches, etc.). A practical workflow is to compute the normalized innovation squared (NIS)

$$\nu_k = y_k - h(\hat{x}_{k|k-1}), \qquad \mathrm{NIS}_k = \nu_k^T S_k^{-1} \nu_k$$

and reject measurements with too-large NIS values. For 2D measurements, a commonly used 99% threshold is approximately `9.21`.

The example below injects two synthetic outliers and compares plain EKF vs. EKF with NIS gating.

Code: EKF with and without NIS gating (tutorial best practice)

```
rng = np.random.default_rng(9)
dyn = create_dynamics("cv", T=0.1, Q=0.08 * np.eye(2), n=2)
obs = create_observation("pos", R=0.25 * np.eye(2), n=2)

N = 80
x0 = np.array([0.0, 0.0, 1.0, 0.6])
truth = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=False)
meas = StateTrajectory.preallocate(state_dim=2, steps=N, with_cov=False)

x = x0
for k in range(N):
    truth.write_mean(k, x)
    yk = obs.random(x=x, rng=rng).ravel()
    if k in (25, 55):
        yk = yk + np.array([4.0, -4.0])  # synthetic outlier
    meas.write_mean(k, yk)
    x = dyn.random(x=x, rng=rng).ravel()

filt = estimator.KalmanFilter(dyn_model=dyn)
state_plain = GaussianState(x=np.zeros(4), P=10 * np.eye(4))
state_gated = GaussianState(x=np.zeros(4), P=10 * np.eye(4))

est_plain = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=True)
est_gated = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=True)

nis_threshold = 9.21  # approx chi^2_{2, 0.99}
accepted = 0
rejected = 0

for k, y in enumerate(meas):
    state_plain = filt.meas_update(state_plain, obs, y)
    est_plain.write_state(k, state_plain)
    state_plain = filt.time_update(state_plain)

    H = obs.H(state_gated.mean())
    S = H @ state_gated.cov() @ H.T + obs.R
    innov = y - obs.h(state_gated.mean())
    nis = float((innov.T @ np.linalg.solve(S, innov)).item())

    if nis < nis_threshold:
        state_gated = filt.meas_update(state_gated, obs, y)
```

```
        accepted += 1
    else:
        rejected += 1

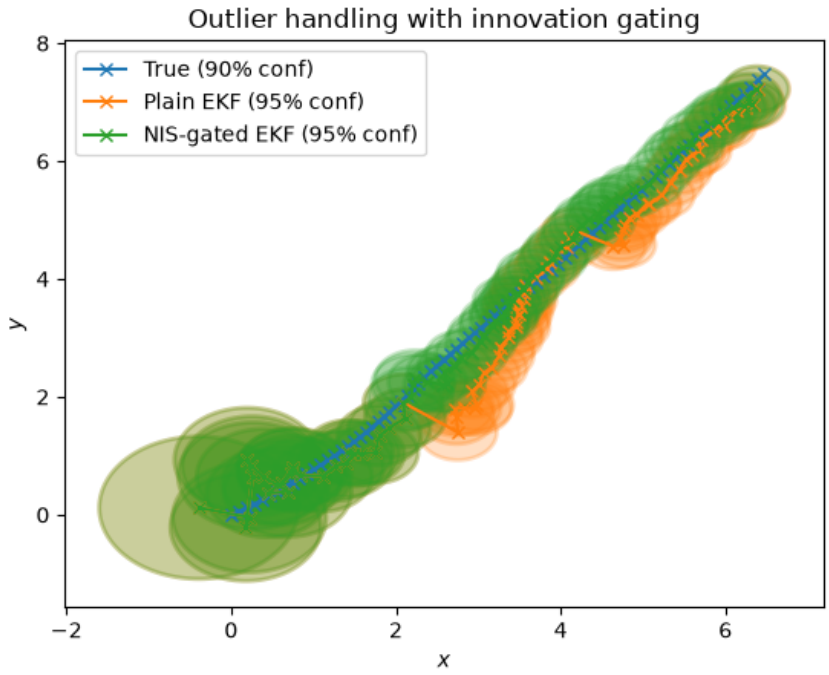
    est_gated.write_state(k, state_gated)
    state_gated = filt.time_update(state_gated)

rmse_plain = np.sqrt(np.mean((est_plain.mean[:2] - truth.mean[:2]) ** 2))
rmse_gated = np.sqrt(np.mean((est_gated.mean[:2] - truth.mean[:2]) ** 2))
print(f"Plain EKF RMSE: {rmse_plain:.3f}")
print(f"Gated EKF RMSE: {rmse_gated:.3f}")
print(f"Accepted measurements: {accepted}, rejected measurements: {rejected}")

fig, ax = plt.subplots()
plot.trajectory(truth, select=(0, 1), label="True", ax=ax)
plot.trajectory(est_plain, select=(0, 1), conf=0.95, label="Plain EKF", ax=ax)
plot.trajectory(est_gated, select=(0, 1), conf=0.95, label="NIS-gated EKF", ax=ax)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("Outlier handling with innovation gating")
ax.legend()
plt.show()
```

Output: EKF with and without NIS gating (tutorial best practice)

```
Plain EKF RMSE: 0.266
Gated EKF RMSE: 0.172
Accepted measurements: 77, rejected measurements: 3
```



5.5 The same loop with UKF and particle filter

The UKF and particle filter use the same two-update interface as the EKF. Create them with `UnscentedKalmanFilter` and `ParticleFilter`, respectively. The UKF keeps the `GaussianState` container; the particle filter works on a `ParticleState`, initialized from a Gaussian prior with `init_state_gaussian(mean, cov, n_particles)`, and needs an explicit `resample(...)` call in the loop. `StateTrajectory.write_state` accepts both containers, so recording estimates is unchanged.

The example reuses the constant-velocity model, now with range (`toa`) measurements to four anchors, which are nonlinear in the position — a case where the three filters genuinely differ in how they propagate uncertainty.

Code: EKF, UKF, and PF on the same range-measurement problem

```
rng = np.random.default_rng(7)
dyn = create_dynamics("cv", T=0.1, Q=0.05 * np.eye(2), n=2)
anchors = np.array([[0.0, 10.0, 10.0, 0.0], [0.0, 0.0, 10.0, 10.0]])
obs = create_observation("toa", p=anchors, R=0.05 * np.eye(4), n=4)

N = 60
truth = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=False)
meas = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=False)
x = np.array([2.0, 2.0, 0.8, 0.6])
for k in range(N):
    truth.write_mean(k, x)
    meas.write_mean(k, obs.random(x=x, rng=rng).ravel())
```

```
    x = dyn.random(x=x, rng=rng).ravel()

x_prior = np.array([3.0, 3.0, 0.0, 0.0])
P_prior = np.diag([4.0, 4.0, 1.0, 1.0])

ekf = estimator.KalmanFilter(dyn_model=dyn)
ukf = estimator.UnscentedKalmanFilter(dyn_model=dyn)
pf = estimator.ParticleFilter(dyn_model=dyn, rng=np.random.default_rng(8))

ekf_state = GaussianState(x=x_prior, P=P_prior)
ukf_state = GaussianState(x=x_prior, P=P_prior)
pf_state = pf.init_state_gaussian(x_prior, P_prior, n_particles=10_000)

est_ekf = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=True)
est_ukf = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=True)
est_pf = StateTrajectory.preallocate(state_dim=4, steps=N, with_cov=True)

for k, y in enumerate(meas):
    ekf_state = ekf.meas_update(ekf_state, obs, y)
    est_ekf.write_state(k, ekf_state)
    ekf_state = ekf.time_update(ekf_state)

    ukf_state = ukf.meas_update(ukf_state, obs, y)
    est_ukf.write_state(k, ukf_state)
    ukf_state = ukf.time_update(ukf_state)

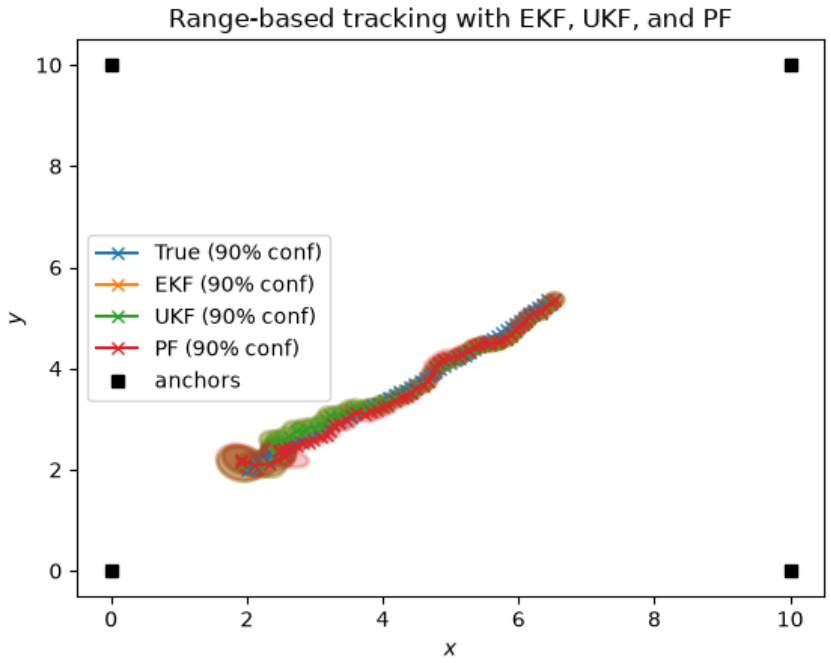
    pf_state = pf.meas_update(pf_state, obs, y)
    est_pf.write_state(k, pf_state)
    pf_state = pf.resample(pf_state)
    pf_state = pf.time_update(pf_state)

for name, est in (("EKF", est_ekf), ("UKF", est_ukf), ("PF ", est_pf)):
    rmse = np.sqrt(np.mean((est.mean[:2] - truth.mean[:2]) ** 2))
    print(f"{name} position RMSE: {rmse:.3f}")

fig, ax = plt.subplots()
plot.trajectory(truth, select=(0, 1), label="True", ax=ax)
plot.trajectory(est_ekf, select=(0, 1), label="EKF", ax=ax)
plot.trajectory(est_ukf, select=(0, 1), label="UKF", ax=ax)
plot.trajectory(est_pf, select=(0, 1), label="PF", ax=ax)
ax.plot(anchors[0], anchors[1], "ks", label="anchors")
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("Range-based tracking with EKF, UKF, and PF")
ax.legend()
plt.show()
```

Output: EKF, UKF, and PF on the same range-measurement problem

```
EKF position RMSE: 0.079
UKF position RMSE: 0.079
PF  position RMSE: 0.083
```



Try it yourself: Filter comparison

1. Shrink the number of particles (10000 → 500) and observe when the PF estimate degrades. With a four-dimensional state and precise range measurements, the particle filter needs many particles to stay healthy.
2. Move the initial prior mean further from the true start and compare how quickly each filter recovers.
3. Replace `toa` with `tdoa`, change `R` to a `(3, 3)` covariance for the three range differences, and repeat the comparison.

6 Moment transforms and analysis helpers

The `helper` module provides moment-transform utilities that propagate a Gaussian belief through a nonlinear function: `ut` (unscented transform, used internally by `UnscentedKalmanFilter`), `tt` (second-order Taylor transform), and `mct` (Monte Carlo transform, a finite-sample reference for

comparing ut/tt). It also provides llh and crlb for evaluating likelihood/CRLB maps over a 2D grid. The crlb helper marks points with undefined Jacobians or singular information as NaN and warns once per call; these entries mean the bound is unavailable, not zero uncertainty.

Below we transform a polar (r, theta) Gaussian belief into Cartesian coordinates using the unscented transform, a first-order Taylor transform, and a 20,000-sample Monte Carlo transform (the reference), and compare the resulting means and variances. For tt, pass the function together with its derivatives: fun=(f, F) for first-order propagation or fun=(f, F, H) to include second-order terms via the output Hessians.

Code: Unscented vs. Taylor vs. Monte Carlo transform

```
def polar_to_cartesian(x):
    r, theta = x[0], x[1]
    return np.stack((r * np.cos(theta), r * np.sin(theta)), axis=0)


def polar_to_cartesian_jacobian(x):
    r, theta = float(x[0, 0]), float(x[1, 0])
    return np.array(
        [[np.cos(theta), -r * np.sin(theta)], [np.sin(theta), r * np.cos(theta)]]
    )


x_polar = GaussianState(x=np.array([10.0, 0.2]), P=np.diag([0.5**2, 0.05**2]))

rng = np.random.default_rng(3)
mc_result = helper.mct(x_polar, polar_to_cartesian, n_samples=20_000, rng=rng)
ut_result = helper.ut(x_polar, polar_to_cartesian)
tt_result = helper.tt(x_polar, (polar_to_cartesian, polar_to_cartesian_jacobian))

print("MC mean:", np.round(mc_result.mean().ravel(), 3),
      " cov diag:", np.round(np.diag(mc_result.cov()), 4))
print("UT mean:", np.round(ut_result.mean().ravel(), 3),
      " cov diag:", np.round(np.diag(ut_result.cov()), 4))
print("TT mean:", np.round(tt_result.mean().ravel(), 3),
      " cov diag:", np.round(np.diag(tt_result.cov()), 4))
```

Output: Unscented vs. Taylor vs. Monte Carlo transform

```
MC mean: [9.788 1.989]  cov diag: [0.2465 0.2502]
UT mean: [9.788 1.984]  cov diag: [0.2503 0.25  ]
TT mean: [9.801 1.987]  cov diag: [0.25 0.25]
```

helper.sigma_points exposes the points and separate mean/covariance weights used by helper.ut. Calling helper.mct(..., return_result=True) returns the estimate and exact input/output samples used for the moments, without extra random draws. These transforms return mean/covariance summaries; the transformed distribution need not be Gaussian.

Try it yourself: Moment-transform accuracy

1. Increase angular uncertainty in `x_polar`.
2. Compare UT and MC means/covariances again.
3. Explain when UT remains accurate and when nonlinear curvature makes MC a better reference.

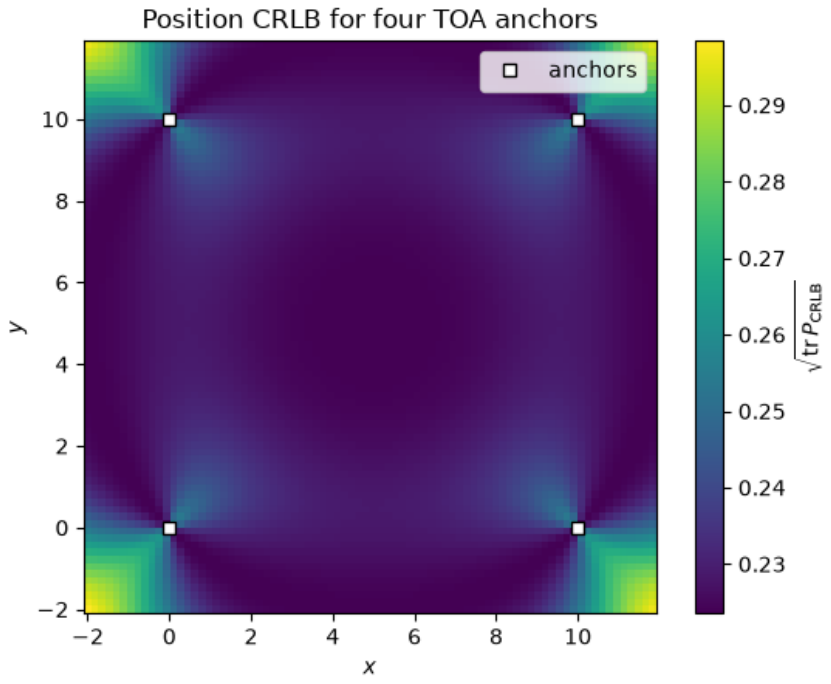
6.1 Analysis helpers: CRLB map

`helper.crlb(model, region, ...)` evaluates a scalar summary (by default the trace) of the Cramer-Rao covariance bound over a 2D position grid — a quick way to inspect the geometry's local bound for unbiased estimation, not a promise of an estimator's actual accuracy. Grid points with undefined Jacobians or singular information become `NaN` (the bound is unavailable there, not zero), and one `RuntimeWarning` per call reports how many points were excluded. `helper.llh(model, y, region, ...)` evaluates a log-likelihood map over the same kind of grid for one measurement.

Code: CRLB map for the TOA anchor geometry

```
toa_obs = create_observation("toa", p=anchors, R=0.05 * np.eye(4), n=4)
(X1, X2), bound = helper.crlb(toa_obs, ((-2.0, 12.0), (-2.0, 12.0)), n_points=80)

fig, ax = plt.subplots()
pcm = ax.pcolormesh(X1, X2, np.sqrt(bound), shading="auto")
fig.colorbar(pcm, ax=ax, label=r"$\sqrt{\operatorname{tr} P_{\mathrm{CRLB}}}$")
ax.plot(anchors[0], anchors[1], "ws", markeredgecolor="k", label="anchors")
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("Position CRLB for four TOA anchors")
ax.set_aspect("equal")
ax.legend()
plt.show()
```



The bound is lowest inside the anchor square and grows outside it, where the range geometry gives less information — exactly the kind of insight to check before trusting an estimator in a given region.

7 Working on exercises

7.1 Participant starter template (recommended workflow)

Copy this sequence into new exercise scripts and fill in model details:

1. Set RNG and constants.
2. Define dynamic + observation models.
3. Simulate a small synthetic case first.
4. Run estimator.
5. Compute one metric (RMSE, bias, acceptance rate, etc.).
6. Plot estimate + uncertainty and compare against ground truth.

Code: Reusable script skeleton for exercises

```
def run_exercise_template(seed: int = 0, steps: int = 40) -> float:
    rng = np.random.default_rng(seed)

    dyn = create_dynamics("cv", T=0.1, Q=0.05 * np.eye(2), n=2)
    obs = create_observation("pos", R=0.4 * np.eye(2), n=2)
    filt = estimator.KalmanFilter(dyn_model=dyn)

    truth = StateTrajectory.preallocate(state_dim=4, steps=steps, with_cov=False)
    est = StateTrajectory.preallocate(state_dim=4, steps=steps, with_cov=True)

    x = np.array([0.0, 0.0, 1.0, 0.3])
    state = GaussianState(x=np.zeros(4), P=8.0 * np.eye(4))

    for k in range(steps):
        truth.write_mean(k, x)
        y = obs.random(x=x, rng=rng)
        state = filt.meas_update(state, obs, y)
        est.write_state(k, state)
        state = filt.time_update(state)
        x = dyn.random(x=x, rng=rng).ravel()

    rmse = float(np.sqrt(np.mean((est.mean[:2] - truth.mean[:2]) ** 2)))
    return rmse


print(f"Template run RMSE: {run_exercise_template(seed=4):.3f}")
```

Output: Reusable script skeleton for exercises

```
Template run RMSE: 0.300
```

7.2 Common pitfalls (and fast fixes)

- **Wrong shape (`(n,)` vs `(n, 1)`)** Fix: call `x.reshape(-1, 1)` before passing state vectors to models.
- **Overconfident `P0` causes divergence** Fix: increase initial covariance and retune process noise `Q`.
- **Model mismatch (wrong sampling time `T`)** Fix: confirm `T` in data loading, model creation, and plotting.
- **Reasonable trajectory but poor uncertainty** Fix: monitor innovation/NIS and compare empirical error against covariance.
- **Non-reproducible results between runs** Fix: use one explicit RNG and pass it through all simulation calls.

7.3 API quick reference

The tutorial sections above explain concepts step by step, while this final section keeps API-oriented reference material for fast lookup. All modules below belong to `sftools`: for example, `types` means `sftools.types`. The package root is listed as `sftools`.

Module	Interfaces / model names
sftools	create_dynamics, create_observation
sftools	register_dynamics, register_observation
types	GaussianState, ParticleState, StateTrajectory
model_dyn	linear, cv, ct
model_obs	linear, nonlinear, pos, toa, tdoa, tdoa_r0
estimator	ls, wls, nls
estimator	KalmanFilter, UnscentedKalmanFilter, ParticleFilter
helper	ut, tt, mct, sigma_points, llh, crlb
plot	estimate, signal, trajectory

This manual shows the essential usage; for full signatures, argument details, and failure modes, use Python's built-in help (for example `help(estimator.nls)`) or the module docstrings.