`sftools` MATLAB Manual

TSRT14 Sensor Fusion

November 21, 2026

Contents

1 Overview **3**

2 Shapes and conventions **3**

3 Suggested learning path **3**

4 State containers **4**

5 Trajectory details **5**
5.1 Try it yourself: State containers 5

6 Dynamic models **5**
6.1 Try it yourself: Dynamic models 7

7 Observation models **7**
7.1 Try it yourself: Observation models 8

8 Least-squares estimation **8**

9 Nonlinear least squares **9**
9.1 Try it yourself: Static localization with NLS 10

10 Recursive filtering **11**
10.1 Try it yourself: Filter tuning 13

11 Innovation monitoring and outlier rejection **13**

12 The same loop with EKF, UKF, and particle filter **15**
12.1 Try it yourself: Filter comparison 16

13 Moment transforms **17**

14 Transform diagnostics **18**
14.1 Try it yourself: Moment-transform accuracy 18

15 Observation-model analysis **18**

16 Working on exercises **19**

17 Common pitfalls **20**

18 API quick reference **20**

About this document

The examples are run when this manual is built, so the displayed output and figures come directly from the accompanying MATLAB script. 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` is a MATLAB toolbox for learning estimation, filtering, and sensor fusion in the TSRT14 Sensor Fusion course at Linköping University. Its teaching-oriented design keeps the structure of each estimation problem visible: for example, you write the filtering loop and call measurement and time updates separately.

The toolbox provides state containers, dynamic and observation models, least-squares estimators, recursive filters, moment transforms, observation-model analysis, and plotting helpers. This manual follows the same learning path as the Python manual, with MATLAB syntax and conventions. It assumes familiarity with the course concepts; it is not a textbook. For complete signatures, use MATLAB's `help` command.

Add the repository root to the MATLAB path before using the toolbox, for example `addpath(pwd)` when the current folder is the repository root. No toolbox is required for the library's known built-in functionality: nonlinear least squares uses Optimization Toolbox when available and otherwise uses a base-MATLAB solver. User-provided models and callbacks may have additional dependencies. The PDF build closes figures after capturing them; ordinary script execution leaves them open.

2 Shapes and conventions

States, measurements, and sample collections are stored in columns. If `n` is the state dimension, `m` the measurement dimension, and `N` the number of samples or time steps, the usual shapes are:

- State: `n-by-1`; measurement: `m-by-1`.
- State samples: `n-by-N`; predicted measurements: `m-by-N`.
- Measurement Jacobian: `m-by-n`; measurement covariance: `m-by-m`.

Estimators, likelihood evaluation, and filter measurement updates require one measurement column. `GaussianState.mean()` returns a column; MATLAB indices are one-based. Model factory arguments are positional, in the same order as in Python.

3 Suggested learning path

Start with state containers and their dimensions, then explore dynamic and observation models, least-squares estimation, recursive filtering, and finally diagnostics and plotting. Keep each

exercise in a saved MATLAB script that runs from start to finish. Use explicit random streams, inspect sizes, choose conservative initial covariances, track an error metric, and plot uncertainty as well as estimates.

4 State containers

`sftools.GaussianState(x, P)` stores a mean and covariance. `sftools.ParticleState(x, logWeights)` stores samples as columns and log weights as a row; its `mean()` and `cov()` return weighted empirical moments. `sftools.StateTrajectory` records state means and optional covariances.

Code: Gaussian and particle states

```
x0 = [1; 2];
P0 = diag([.5, .2]);
state = sftools.GaussianState(x0, P0);
disp('Gaussian mean:'); disp(state.mean());
disp('Gaussian covariance:'); disp(state.cov());

stream = RandStream('mt19937ar', 'Seed', 0);
particles = x0 + chol(P0, 'lower')*randn(stream, 2, 500);
particleState = sftools.ParticleState(particles, -log(500)*ones(1, 500));
disp('Particle mean:'); disp(particleState.mean());
disp('Particle covariance:'); disp(particleState.cov());
```

Output: Gaussian and particle states

```
Gaussian mean:
     1
     2

Gaussian covariance:
    0.5000         0
         0    0.2000

Particle mean:
    0.9925
    1.9756

Particle covariance:
    0.5020    0.0162
    0.0162    0.1978
```

Code: State trajectories

```
traj = sftools.StateTrajectory.preallocate(2, 5, true);
for k = 1:5
  traj.writeState(k, sftools.GaussianState(x0 + k - 1, P0));
end
disp('Trajectory dimensions:'); disp(size(traj.mean));
disp('First component:'); disp(traj.component(1));
iterator = traj.sampleIterator();
disp('First sample:'); disp(iterator.next());
```

Output: State trajectories

```
Trajectory dimensions:
     2     5

First component:
     1     2     3     4     5

First sample:
     1
     2
```

5 Trajectory details

`preallocateLike(other, stateDim, withCovariance)` preserves the number of steps and time stamps of another trajectory. `iterSamples` returns a cell array; `sampleIterator` traverses lazily using `hasNext()` and `next()`. Optional covariance and time metadata can be requested from either method. Trajectories are handle objects, so writing a sample changes the object.

5.1 Try it yourself: State containers

1. Increase one diagonal entry of `P0` and inspect the particle covariance.
2. Increase the number of particles and compare the empirical moments.
3. Preallocate a longer trajectory and inspect its second component.

6 Dynamic models

Construct a model with `sftools.createDynamicModel`.

Alternatively, use a class in `sftools.models.dynamic`. Each model provides `f(x)`, its Jacobian `F(x)`, noise gain `G(x)`, and `random(n, x, stream)` for stochastic propagation.

- `linear`: user-defined state layout; arguments `F, G, Q`.

- cv: positions followed by velocities; arguments T, Q, n.
- ct: [x; y; speed; heading; turn_rate]; arguments T, Q, n.

T is the sampling interval and Q is process-noise covariance. The added state covariance is G(x)*Q*G(x).'. In a two-dimensional CV model, Q is 2-by-2 even though the state has four components.

Code: Constant-velocity model

```
dyn = sftools.createDynamicModel('cv', .1, .01*eye(2), 2);
x0cv = [0; 0; 1; .5];
disp('f(x0):'); disp(dyn.f(x0cv));
disp('F(x0):'); disp(dyn.F(x0cv));
stream = RandStream('mt19937ar', 'Seed', 1);
disp('Three stochastic samples:'); disp(dyn.random(3, x0cv, stream));
```

Output: Constant-velocity model

```
f(x0):
    0.1000
    0.0500
    1.0000
    0.5000

F(x0):
    1.0000         0    0.1000         0
         0    1.0000         0    0.1000
         0         0    1.0000         0
         0         0         0    1.0000

Three stochastic samples:
    0.0997    0.0996    0.0996
    0.0506    0.0494    0.0497
    0.9935    0.9924    0.9915
    0.5118    0.4889    0.4943
```

Code: Coordinated-turn model

```
ct = sftools.createDynamicModel('ct', .5, diag([.01, .001]), 2);
xc0 = [0; 0; 1; 0; .3];
disp('CT f(x0):'); disp(ct.f(xc0));
```

Output: Coordinated-turn model

```
CT f(x0):
    0.4981
    0.0374
    1.0000
    0.1500
    0.3000
```

6.1 Try it yourself: Dynamic models

1. Change the sampling interval and inspect the transition matrix.
2. Vary `Q` and compare the spread of stochastic samples.
3. Vary the turn rate and observe the coordinated-turn prediction.

7 Observation models

Construct a model with `sftools.createObservationModel(name, ...)` or its class in `sftools.models.observation`. Models provide `h(x)`, `H(x)`, `random(x, n, stream)`, and covariance `R`. The diagonal entries of `R` are variances, not standard deviations.

- `linear`: linear measurement; arguments `H, R`.
- `nonlinear`: supplied mapping and Jacobian; arguments `h, H, R, ...`.
- `pos`: position measurement; arguments `R, n`.
- `toa`: ranges to anchors; arguments `p, R, n`.
- `tdoa`: range differences relative to the first anchor; arguments `p, R, n`.
- `tdoa_r0`: ranges with a shared unknown offset; arguments `p, R, n`.

Anchor positions are columns of `p`. With `n` anchors, `toa` has `n` measurements and `tdoa` has `n-1`. For `tdoa_r0`, the first state components are `[x; y; r0]` and any additional components have zero Jacobian columns. Custom `h` callbacks accept batches of state columns, while `H` evaluates one state. Range Jacobians are undefined at an anchor, so choose linearization points and initial guesses away from the anchor locations.

Code: Time-of-arrival observation model

```
anchors = [0, 10, 10, 0; 0, 0, 10, 10];
obs = sftools.createObservationModel('toa', anchors, .01*eye(4), 4);
xtrue = [3; 4];
disp('h(x):'); disp(obs.h(xtrue));
```

```
disp('H(x):'); disp(obs.H(xtrue));
stream = RandStream('mt19937ar', 'Seed', 2);
y = obs.random(xtrue, 1, stream);
disp('Noisy measurement:'); disp(y);
posObs = sftools.createObservationModel('pos', .1*eye(2), 2);
disp('Position model h(x):'); disp(posObs.h(xtrue));
```

Output: Time-of-arrival observation model

```
h(x):
    5.0000
    8.0623
    9.2195
    6.7082

H(x):
    0.6000    0.8000
   -0.8682    0.4961
   -0.7593   -0.6508
    0.4472   -0.8944

Noisy measurement:
    4.9876
    7.8081
    9.2473
    6.6886

Position model h(x):
     3
     4
```

7.1 Try it yourself: Observation models

1. Move an anchor and inspect the measurements and Jacobian.
2. Increase `R` and compare the noise in generated measurements.
3. Replace `toa` with `tdoa` and compare symmetric and asymmetric geometries.

8 Least-squares estimation

`sftools.estimators.ls` and `wls` return a `GaussianState`. `ls` uses an unweighted linear fit; `wls` accounts for measurement covariance. Both use the design matrix `model.H(x0)`, with optional `x0` passed only to the Jacobian, not used as a nonlinear correction. Their covariances can differ with unequal or correlated noise. Use `nls` for a nonlinear observation model.

Code: LS and WLS with a linear observation model

```
H = [1, 0; 0, 1; 1, 1];
R = [.04, .02, 0; .02, .09, 0; 0, 0, .01];
linearObs = sftools.createObservationModel('linear', H, R);
stream = RandStream('mt19937ar', 'Seed', 39);
xtrue = [1; 1];
y = linearObs.random(xtrue, 1, stream);
xhatLS = sftools.estimators.ls(linearObs, y);
xhatWLS = sftools.estimators.wls(linearObs, y);
disp('LS estimate:'); disp(xhatLS.mean());
disp('WLS estimate:'); disp(xhatWLS.mean());
fig = figure;
ax = axes(fig); hold(ax, 'on');
sftools.plotting.estimate(xhatLS, 'Axes', ax, 'Confidence', .95, 'Label', 'LS');
sftools.plotting.estimate(xhatWLS, 'Axes', ax, 'Confidence', .95, 'Label', 'WLS');
plot(ax, xtrue(1), xtrue(2), 'k*', 'DisplayName', 'True');
xlabel(ax, 'x_1'); ylabel(ax, 'x_2');
title(ax, 'LS and WLS with 95% confidence ellipses');
axis(ax, 'equal'); legend(ax, 'show');
snapnow;
if exist('sftoolsManualPublishing', 'var') && sftoolsManualPublishing, close(fig); end
```

Output: LS and WLS with a linear observation model

```
LS estimate:
    0.8013
    1.2630

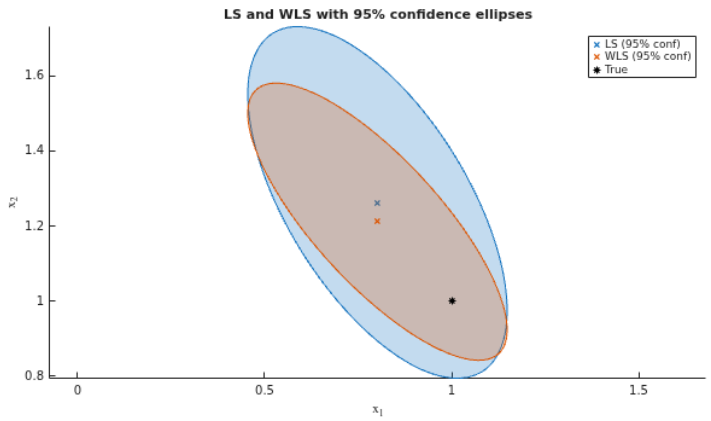
WLS estimate:
    0.8013
    1.2117
```

9 Nonlinear least squares

`nls` evaluates the measurement model and analytic Jacobian from a supplied initial guess. The covariance is the inverse information matrix formed from the whitened residual Jacobian. The local confidence ellipse does not guarantee global optimality or confidence-region coverage. The second output contains diagnostics; on nonconvergence or singular covariance it preserves diagnostics and returns an empty estimate. With one output, those failures raise errors.

Code: NLS range localization

```
rangeObs = sftools.createObservationModel('toa', anchors, .01*eye(4), 4);
xtrue = [3; 4];
stream = RandStream('mt19937ar', 'Seed', 2);
```




```
y = rangeObs.random(xtrue, 1, stream);
[xhatNLS, optimizer] = sftools.estimators.nls(rangeObs, y, [5; 5]);
disp('NLS estimate:'); disp(xhatNLS.mean());
fprintf('Converged: %d; cost: %.4f\n', optimizer.success, optimizer.cost);
fig = figure;
ax = axes(fig); hold(ax, 'on');
sftools.plotting.estimate(xhatNLS, 'Axes', ax, 'Confidence', .95, 'Label', 'NLS');
plot(ax, xtrue(1), xtrue(2), 'k*', 'DisplayName', 'True');
xlabel(ax, 'Position x'); ylabel(ax, 'Position y');
title(ax, 'Range localization'); axis(ax, 'equal'); legend(ax, 'show');
snapnow;
if exist('sftoolsManualPublishing', 'var') && sftoolsManualPublishing, close(fig); end
```

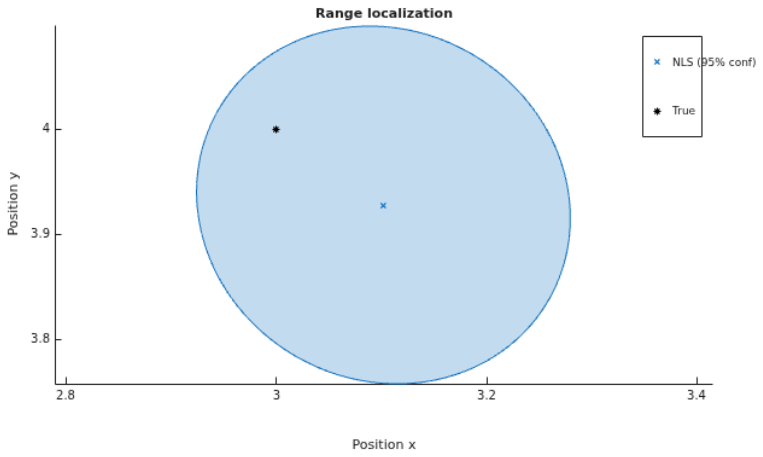
Output: NLS range localization

```
NLS estimate:
    3.1019
    3.9278

Converged: 1; cost: 1.8675
```

9.1 Try it yourself: Static localization with NLS

1. Simulate measurements using different anchor geometries.
2. Run `nls` from two initial guesses.
3. Compare convergence and covariance, and discuss geometry and initialization.

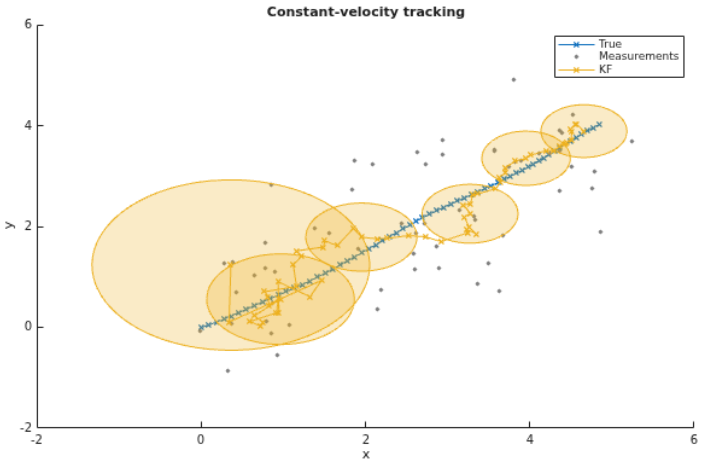


10 Recursive filtering

`KalmanFilter` handles linear models, `ExtendedKalmanFilter` handles nonlinear models, `UnscentedKalmanFilter` propagates sigma points, and `ParticleFilter` implements a bootstrap particle filter. Each has `measurementUpdate(state, obs, y)` and `timeUpdate(state)`. Writing the loop makes correction and prediction explicit and allows individual observations to be inspected or rejected. For a CV state `[x; y; vx; vy]`, use `Select [1, 2]` for positions and `[3, 4]` for velocities.

Code: Kalman filter tracking example

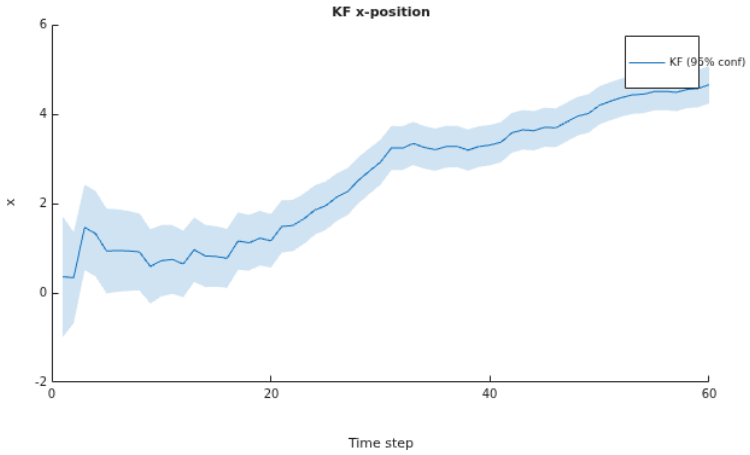
```
stream = RandStream('mt19937ar', 'Seed', 0);
dyn = sftools.createDynamicModel('cv', .1, .1*eye(2), 2);
obs = sftools.createObservationModel('pos', .5*eye(2), 2);
N = 60;
truth = sftools.StateTrajectory.preallocate(4, N, false);
meas = sftools.StateTrajectory.preallocate(2, N, false);
x = [0; 0; 1; .5];
for k = 1:N
  truth.writeMean(k, x);
  meas.writeMean(k, obs.random(x, 1, stream));
  x = dyn.random(1, x, stream);
end
filter = sftools.filters.KalmanFilter(dyn);
state = sftools.GaussianState(zeros(4, 1), 10*eye(4));
est = sftools.StateTrajectory.preallocateLike(meas, 4, true);
for k = 1:N
  state = filter.measurementUpdate(state, obs, meas.stateAt(k));
  est.writeState(k, state);
  state = filter.timeUpdate(state);
end
```




```
rmse = sqrt(mean((est.mean(1:2, :) - truth.mean(1:2, :)).^2, 'all'));
fprintf('Position RMSE: %.3f\n', rmse);
fig = figure;
ax = axes(fig); hold(ax, 'on');
ax.Toolbar.Visible = 'off';
sftools.plotting.trajectory(truth, 'Axes', ax, 'Confidence', 0, 'Label', 'True');
plot(ax, meas.mean(1, :), meas.mean(2, :), '.', 'MarkerSize', 10, ...
  'Color', [.5, .5, .5], 'DisplayName', 'Measurements');
estimateLine = sftools.plotting.trajectory(est, 'Axes', ax, ...
  'Confidence', 0, 'Label', 'KF');
selectedSteps = unique(round(linspace(1, N, 6)));
for k = selectedSteps
  stateEstimate = sftools.GaussianState(est.mean(:, k), est.cov(:, :, k));
  estimateHandle = sftools.plotting.estimate(stateEstimate, ...
    'Axes', ax, 'Confidence', .95, 'Color', estimateLine.Color, ...
    'Label', 'KF confidence');
  set(estimateHandle, 'HandleVisibility', 'off');
end
xlabel(ax, 'x'); ylabel(ax, 'y');
title(ax, 'Constant-velocity tracking'); legend(ax, 'show');
snapnow;
if exist('sftoolsManualPublishing', 'var') && sftoolsManualPublishing, close(fig); end
```

Output: Kalman filter tracking example

```
Position RMSE: 0.339
```



10.1 Try it yourself: Filter tuning

Tune under-modelled, over-modelled, and balanced process noise on the same synthetic trajectory. Compare RMSE and confidence bounds; a plausible trajectory can still be overconfident.

Code: Signal plot of the x-position track

```
fig = figure;
ax = axes(fig);
sftools.plotting.signal(est, 'Axes', ax, 'Select', 1, ...
  'Confidence', .95, 'Label', 'KF');
xlabel(ax, 'Time step'); ylabel(ax, 'x');
title(ax, 'KF x-position'); legend(ax, 'show');
snapnow;
if exist('sftoolsManualPublishing', 'var') && sftoolsManualPublishing, close(fig); end
```

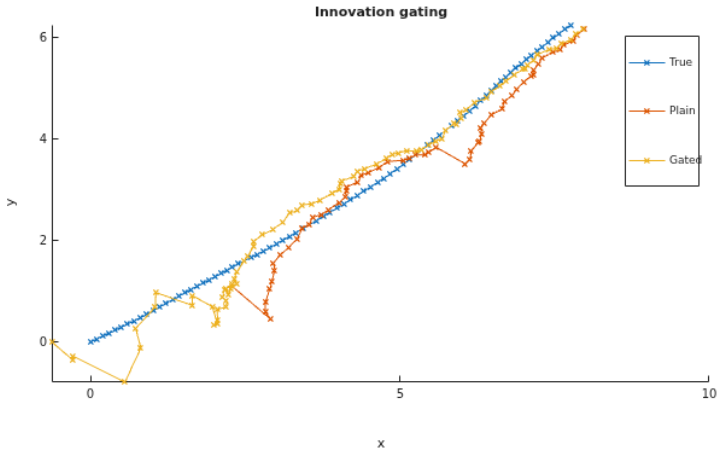
11 Innovation monitoring and outlier rejection

Real measurements can contain gross errors. The normalized innovation squared (NIS) is the innovation transposed times the inverse innovation covariance times the innovation. A commonly used 99 percent threshold for two-dimensional measurements is approximately 9.21. The following example injects two outliers and compares plain filtering with NIS gating.

Code: Filtering with and without NIS gating

```
stream = RandStream('mt19937ar', 'Seed', 9);
dyn = sftools.createDynamicModel('cv', .1, .08*eye(2), 2);
obs = sftools.createObservationModel('pos', .25*eye(2), 2);
N = 80;
```

```
truth = sftools.StateTrajectory.preallocate(4, N, false);
meas = sftools.StateTrajectory.preallocate(2, N, false);
x = [0; 0; 1; .6];
for k = 1:N
  truth.writeMean(k, x);
  yk = obs.random(x, 1, stream);
  if k == 26 || k == 56
    yk = yk + [4; -4];
  end
  meas.writeMean(k, yk);
  x = dyn.random(1, x, stream);
end
filter = sftools.filters.KalmanFilter(dyn);
plain = sftools.GaussianState(zeros(4, 1), 10*eye(4));
gated = sftools.GaussianState(zeros(4, 1), 10*eye(4));
estPlain = sftools.StateTrajectory.preallocate(4, N, true);
estGated = sftools.StateTrajectory.preallocate(4, N, true);
accepted = 0; rejected = 0;
for k = 1:N
  y = meas.stateAt(k);
  plain = filter.measurementUpdate(plain, obs, y);
  estPlain.writeState(k, plain);
  plain = filter.timeUpdate(plain);
  H = obs.H(gated.mean());
  S = H*gated.cov()*H.' + obs.R;
  innovation = y - obs.h(gated.mean());
  nis = innovation.'*(S\innovation);
  if nis < 9.21
    gated = filter.measurementUpdate(gated, obs, y);
    accepted = accepted + 1;
  else
    rejected = rejected + 1;
  end
  estGated.writeState(k, gated);
  gated = filter.timeUpdate(gated);
end
rmsePlain = sqrt(mean((estPlain.mean(1:2, :) - truth.mean(1:2, :)).^2, 'all'));
rmseGated = sqrt(mean((estGated.mean(1:2, :) - truth.mean(1:2, :)).^2, 'all'));
fprintf('Plain RMSE: %.3f; gated RMSE: %.3f\n', rmsePlain, rmseGated);
fprintf('Accepted: %d; rejected: %d\n', accepted, rejected);
fig = figure;
ax = axes(fig); hold(ax, 'on');
sftools.plotting.trajectory(truth, 'Axes', ax, 'Confidence', 0, 'Label', 'True');
sftools.plotting.trajectory(estPlain, 'Axes', ax, 'Confidence', 0, 'Label', 'Plain');
sftools.plotting.trajectory(estGated, 'Axes', ax, 'Confidence', 0, 'Label', 'Gated');
xlabel(ax, 'x'); ylabel(ax, 'y'); title(ax, 'Innovation gating');
legend(ax, 'show');
snapnow;
if exist('sftoolsManualPublishing', 'var') && sftoolsManualPublishing, close(fig); end
```



Output: Filtering with and without NIS gating

```
Plain RMSE: 0.355; gated RMSE: 0.303
Accepted: 77; rejected: 3
```

12 The same loop with EKF, UKF, and particle filter

The filters share the two-update interface. The EKF and UKF use Gaussian states; the particle filter is initialized with `initStateGaussian` and needs an explicit `resample` call. This example uses nonlinear range measurements so the three filters differ in how they propagate uncertainty.

Code: Range-based filter comparison

```
stream = RandStream('mt19937ar', 'Seed', 7);
dyn = sftools.createDynamicModel('cv', .1, .05*eye(2), 2);
anchors = [0, 10, 10, 0; 0, 0, 10, 10];
obs = sftools.createObservationModel('toa', anchors, .05*eye(4), 4);
N = 60;
truth = sftools.StateTrajectory.preallocate(4, N, false);
meas = sftools.StateTrajectory.preallocate(4, N, false);
x = [2; 2; .8; .6];
for k = 1:N
  truth.writeMean(k, x);
  meas.writeMean(k, obs.random(x, 1, stream));
  x = dyn.random(1, x, stream);
end
priorMean = [3; 3; 0; 0];
priorCov = diag([4, 4, 1, 1]);
ekf = sftools.filters.ExtendedKalmanFilter(dyn);
```

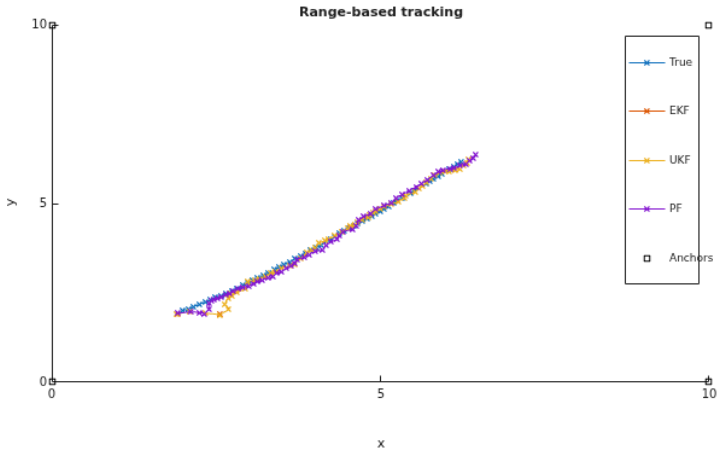
```
ukf = sftools.filters.UnscentedKalmanFilter(dyn);
pf = sftools.filters.ParticleFilter(dyn, RandStream('mt19937ar', 'Seed', 8));
ekfState = sftools.GaussianState(priorMean, priorCov);
ukfState = sftools.GaussianState(priorMean, priorCov);
pfState = pf.initStateGaussian(priorMean, priorCov, 1500);
estEKF = sftools.StateTrajectory.preallocate(4, N, true);
estUKF = sftools.StateTrajectory.preallocate(4, N, true);
estPF = sftools.StateTrajectory.preallocate(4, N, true);
for k = 1:N
  y = meas.stateAt(k);
  ekfState = ekf.measurementUpdate(ekfState, obs, y);
  estEKF.writeState(k, ekfState);
  ekfState = ekf.timeUpdate(ekfState);
  ukfState = ukf.measurementUpdate(ukfState, obs, y);
  estUKF.writeState(k, ukfState);
  ukfState = ukf.timeUpdate(ukfState);
  pfState = pf.measurementUpdate(pfState, obs, y);
  estPF.writeState(k, pfState);
  pfState = pf.timeUpdate(pf.resample(pfState));
end
rmseEKF = sqrt(mean((estEKF.mean(1:2, :) - truth.mean(1:2, :)).^2, 'all'));
rmseUKF = sqrt(mean((estUKF.mean(1:2, :) - truth.mean(1:2, :)).^2, 'all'));
rmsePF = sqrt(mean((estPF.mean(1:2, :) - truth.mean(1:2, :)).^2, 'all'));
fprintf('Position RMSE: EKF %.3f, UKF %.3f, PF %.3f\n', rmseEKF, rmseUKF, rmsePF);
fig = figure;
ax = axes(fig); hold(ax, 'on');
sftools.plotting.trajectory(truth, 'Axes', ax, 'Confidence', 0, 'Label', 'True');
sftools.plotting.trajectory(estEKF, 'Axes', ax, 'Confidence', 0, 'Label', 'EKF');
sftools.plotting.trajectory(estUKF, 'Axes', ax, 'Confidence', 0, 'Label', 'UKF');
sftools.plotting.trajectory(estPF, 'Axes', ax, 'Confidence', 0, 'Label', 'PF');
plot(ax, anchors(1, :), anchors(2, :), 'ks', 'DisplayName', 'Anchors');
xlabel(ax, 'x'); ylabel(ax, 'y'); title(ax, 'Range-based tracking');
legend(ax, 'show');
snapnow;
if exist('sftoolsManualPublishing', 'var') && sftoolsManualPublishing, close(fig); end
```

Output: Range-based filter comparison

```
Position RMSE: EKF 0.091, UKF 0.091, PF 0.245
```

12.1 Try it yourself: Filter comparison

1. Reduce the number of particles and observe when the PF estimate degrades.
2. Move the prior mean farther from the true starting state.
3. Replace `toa` with `tdoa` and use a `3-by-3` measurement covariance.



13 Moment transforms

`sftools.transforms.ut`, `tt`, and `mct` propagate a Gaussian belief through a nonlinear mapping using sigma points, a Taylor approximation, and Monte Carlo samples respectively. `tt` accepts a cell array `{f, F}` for first order or `{f, F, H}` for second order; output Hessians use `state-by-state-by-output` ordering. All transformations return a Gaussian summary, which does not imply that the transformed distribution is Gaussian.

Code: Unscented, Taylor, and Monte Carlo transforms

```
polarToCartesian = @(x) [x(1, :).*cos(x(2, :)); x(1, :).*sin(x(2, :))];
polarJacobian = @(x) [cos(x(2)), -x(1)*sin(x(2)); ...
  sin(x(2)), x(1)*cos(x(2))];
polar = sftools.GaussianState([10; .2], diag([.5^2, .05^2]));
stream = RandStream('mt19937ar', 'Seed', 3);
mc = sftools.transforms.mct(polar, polarToCartesian, 20000, stream);
ut = sftools.transforms.ut(polar, polarToCartesian);
tt = sftools.transforms.tt(polar, {polarToCartesian, polarJacobian});
disp('Monte Carlo mean and covariance:'); disp(mc.mean()); disp(mc.cov());
disp('Unscented mean and covariance:'); disp(ut.mean()); disp(ut.cov());
disp('Taylor mean and covariance:'); disp(tt.mean()); disp(tt.cov());
```

Output: Unscented, Taylor, and Monte Carlo transforms

```
Monte Carlo mean and covariance:
    9.7851
    1.9893

    0.2524    0.0009
    0.0009    0.2508
```

```
Unscented mean and covariance:
    9.7884
    1.9842

    0.2503    0.0001
    0.0001    0.2500

Taylor mean and covariance:
    9.8007
    1.9867

    0.2500   -0.0000
   -0.0000    0.2500
```

14 Transform diagnostics

`sigmaPoints` returns points and separate mean and covariance weights. Additional outputs from `ut` retain input and output sigma points; extra outputs from `mct` retain the exact input and output samples without additional random draws. Monte Carlo callbacks must return one output column per sample and need at least two samples.

14.1 Try it yourself: Moment-transform accuracy

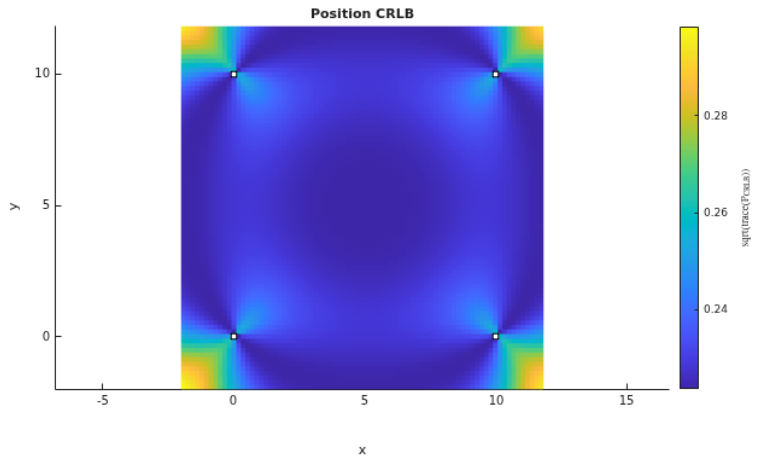
Increase angular uncertainty and compare unscented and Monte Carlo results. Explain when nonlinear curvature makes Monte Carlo a better reference.

15 Observation-model analysis

`sftools.analysis.crlb` evaluates a scalar summary of the Cramer-Rao covariance bound over a two-dimensional grid. The default summary is the trace; the bound describes local geometry for unbiased estimation, not guaranteed estimator accuracy. Undefined or singular points become `NaN` and produce a warning. `sftools.analysis.llh` evaluates a log-likelihood map for one measurement. Bounds are `[xmin, xmax; ymin, ymax]` and upper limits are excluded.

Code: CRLB map for the TOA anchor geometry

```
toaObs = sftools.createObservationModel('toa', anchors, .05*eye(4), 4);
[grid, bound] = sftools.analysis.crlb(toaObs, [-2, 12; -2, 12], 80);
fig = figure;
ax = axes(fig); hold(ax, 'on');
surf(ax, grid{1}, grid{2}, sqrt(bound), 'EdgeColor', 'none');
view(ax, 2);
bar = colorbar(ax); bar.Label.String = 'sqrt(trace(P_{CRLB}))';
z = max(sqrt(bound(:)), [], 'omitnan') + 1;
plot3(ax, anchors(1, :), anchors(2, :), z*ones(1, size(anchors, 2)), ...
```




```
  'ws', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'w', 'MarkerSize', 6);
xlabel(ax, 'x'); ylabel(ax, 'y'); title(ax, 'Position CRLB');
axis(ax, 'equal');
snapnow;
if exist('sftoolsManualPublishing', 'var') && sftoolsManualPublishing, close(fig); end
```

16 Working on exercises

A useful starting workflow is to initialize an explicit random stream, define models, simulate a small case, run estimators, compute an error metric, and plot estimates and uncertainty against ground truth.

Code: Reusable exercise skeleton

```
stream = RandStream('mt19937ar', 'Seed', 4);
steps = 40;
dyn = sftools.createDynamicModel('cv', .1, .05*eye(2), 2);
obs = sftools.createObservationModel('pos', .4*eye(2), 2);
filter = sftools.filters.KalmanFilter(dyn);
truth = sftools.StateTrajectory.preallocate(4, steps, false);
est = sftools.StateTrajectory.preallocate(4, steps, true);
x = [0; 0; 1; .3];
state = sftools.GaussianState(zeros(4, 1), 8*eye(4));
for k = 1:steps
  truth.writeMean(k, x);
  y = obs.random(x, 1, stream);
  state = filter.measurementUpdate(state, obs, y);
  est.writeState(k, state);
  state = filter.timeUpdate(state);
```

```
  x = dyn.random(1, x, stream);
end
rmse = sqrt(mean((est.mean(1:2, :) - truth.mean(1:2, :)).^2, 'all'));
fprintf('Template run RMSE: %.3f\n', rmse);
```

Output: Reusable exercise skeleton

```
Template run RMSE: 0.312
```

17 Common pitfalls

- Use explicit column vectors for measurements and states.
- Increase overconfident initial covariance and retune process noise.
- Match sampling time to the data and model.
- Check innovations and covariance, not only the plotted trajectory.
- Pass explicit streams through simulations for reproducibility. MATLAB and NumPy do not generate identical random sequences.

18 API quick reference

- `sftools`: state containers, model factories, and registration functions.
- `sftools.models.dynamic`: linear, CV, and coordinated-turn models.
- `sftools.models.observation`: linear, nonlinear, position, TOA, TDOA, and offset-range models.
- `sftools.estimators`: `ls`, `wls`, and `nls`.
- `sftools.filters`: Kalman, extended Kalman, unscented Kalman, and particle filters.
- `sftools.transforms`: `ut`, `tt`, `mct`, and `sigmaPoints`.
- `sftools.analysis`: `llh` and `crlb`.
- `sftools.plotting`: `estimate`, `signal`, and `trajectory`.

Use `help sftools.estimators.nls` and the corresponding class or function help for full signatures and failure modes.