The starman library

Release 0.0.1

Rich Wareham

April 04, 2016

Contents

CHAPTER 1

Introduction

The starman library provides implementation of algorithms commonly used to estimate state and to track targets over time in the presence of noisy measurements.

Those wanting to dive in and see what is supported may take a look at the Programmer's Reference for all the gory details.

1.1 Features

Starman provides a Kalman filter implementation which can be used to track the hidden true state of a linear system over time given zero or more noisy measurements for each time step.

A Rauch-Tung-Striebel smoother (RTS) implementation is provided which, when combined with the Kalman filter, can produce very smooth estimates of state. Unlike the Kalman filter which provides an estimate of state for each time step based only on measurements up until that time step, the RTS smoother requires all measurements to have been recorded.

For associating multiple measurements per frame to multiple targets, an implementation of the Scott and Longuet-Higgins feature association algorithm is provided. This algorithm can be used to "join the dots" when tracking multiple targets.

1.2 Why "starman"?

Starman implements the Kalman filter. The Kalman filter was used for trajectory estimation in the Apollo spaceflight programme. Starman is thus a blend of "star", signifying space, and "Kalman". That and "kalman" was already taken as a package name on the PyPI.

CHAPTER 2

State estimation

It is often easy enough to write down equations determining the dynamics of a system: how its state varies over time. Given a system at time *k* we can predict what state it will be in at time *k+1*. We can also take measurements on the system at time *k+1*. The process of fusing zero or measurements of a system with predictions of its state is called *state estimation*.

2.1 The Kalman filter

A very popular state estimation algorithm is the Kalman filter. The Kalman filter can be used when the dynamics of a system are linear and measurements are some linear function of state.

2.1.1 Problem formulation

Let's first refresh the goal of the Kalman filter and its formulation. The Kalman filter attempts to update an estimate of the "true" state of a system given noisy measurements of the state. The state is assumed to evolve in a linear way and measurements are assumed to be linear functions of the state. Specifically, it is assumed that the "true" state at time $k+1$ is a function of the "true" state at time k:

$$x_{k+1} = F_k x_k + B_k u_k + w_k$$

where w_k is a sample from a zero-mean Gaussian process with covariance Q_k. We term Q_k the *process covariance*. The matrix F_k is termed the *state-transition matrix* and determines how the state evolves. The matrix B_k is the *control matrix* and determines the contribution to the state of the control input, u_k.

At time instant k we may have zero or more *measurements* of the state. Each measurement, z_k is assumed to be a linear function of the state:

$$z_k = H_k x_k + v_k$$

where H_k is termed the *measurement matrix* and v_k is a sample from a zero-mean Gaussian process with covariance R_k. We term R_k the *measurement covariance*.

The Kalman filter maintains for time instant, k, an *a priori* estimate of state, $\hat{x}_{k|k-1}$ covariance of this estimate, $P_{k|k-1}$. The initial values of these parameters are given when the Kalman filter is created. The filter also maintains an *a posteriori* estimate of state, $\hat{x}_{k|k}$, and covariance, $P_{k|k}$. This is updated for each measurement, z_k.

2.1.2 Example: the constant velocity model

The Kalman filter is implemented in Starman in the `starman.KalmanFilter` class. This section provides an example of use.

Generating the true states

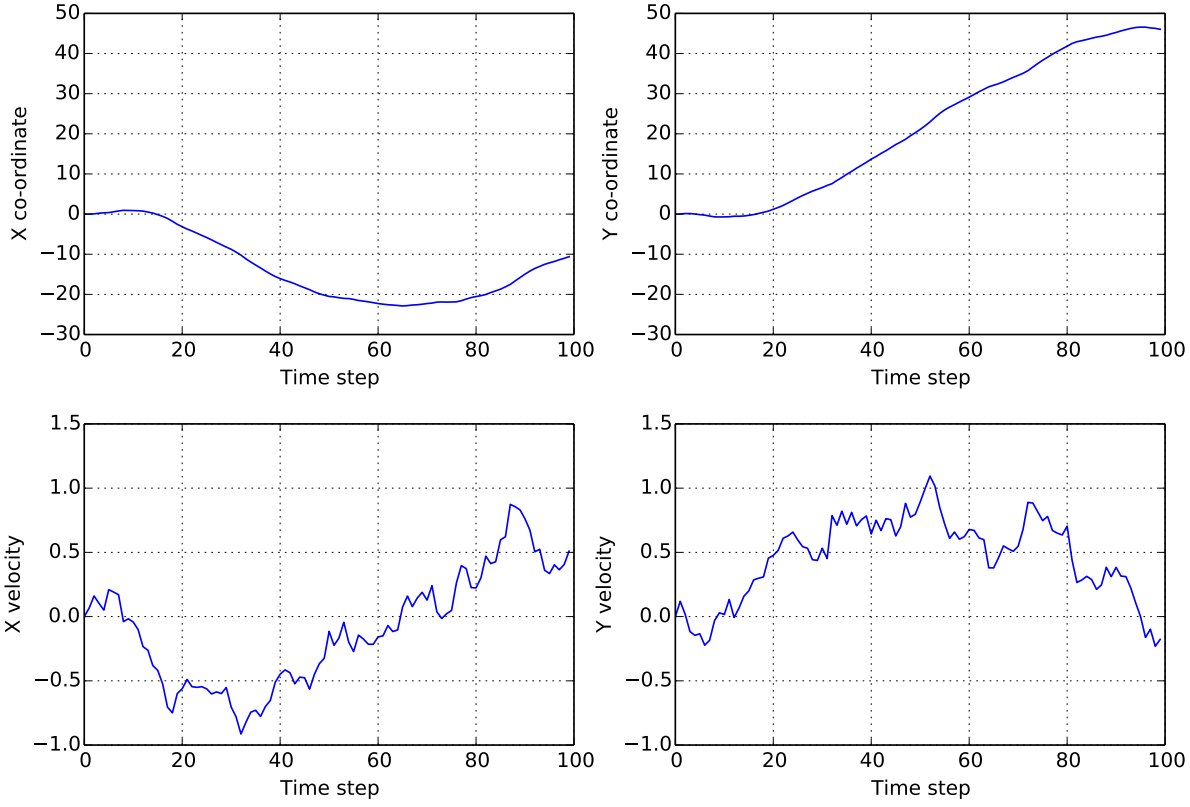
We will implement a simple 2D state estimation problem using the constant velocity model. The state transition matrix is constant throughout the model:

```
# Import numpy and matplotlib functions into global namespace
from matplotlib.pylab import *

# Our state is x-position, y-position, x-velocity and y-velocity.
# The state evolves by adding the corresponding velocities to the
# x- and y-positions.
F = array([
    [1, 0, 1, 0], # x <- x + vx
    [0, 1, 0, 1], # y <- y + vy
    [0, 0, 1, 0], # vx is constant
    [0, 0, 0, 1], # vy is constant
])

# Specify the length of the state vector
STATE_DIM = F.shape[0]
```

Let's generate some sample data by determining the process noise covariance:

```
# Specify the process noise covariance
Q = diag([1e-2, 1e-2, 1e-1, 1e-1]) ** 2

# How many states should we generate?
N = 100

# Generate some "true" states
from starman.linearsystem import generate_states
true_states = generate_states(N, process_matrix=F, process_covariance=Q)
assert true_states.shape == (N, STATE_DIM)
```

We can plot the true states we've just generated:



Generating measurements

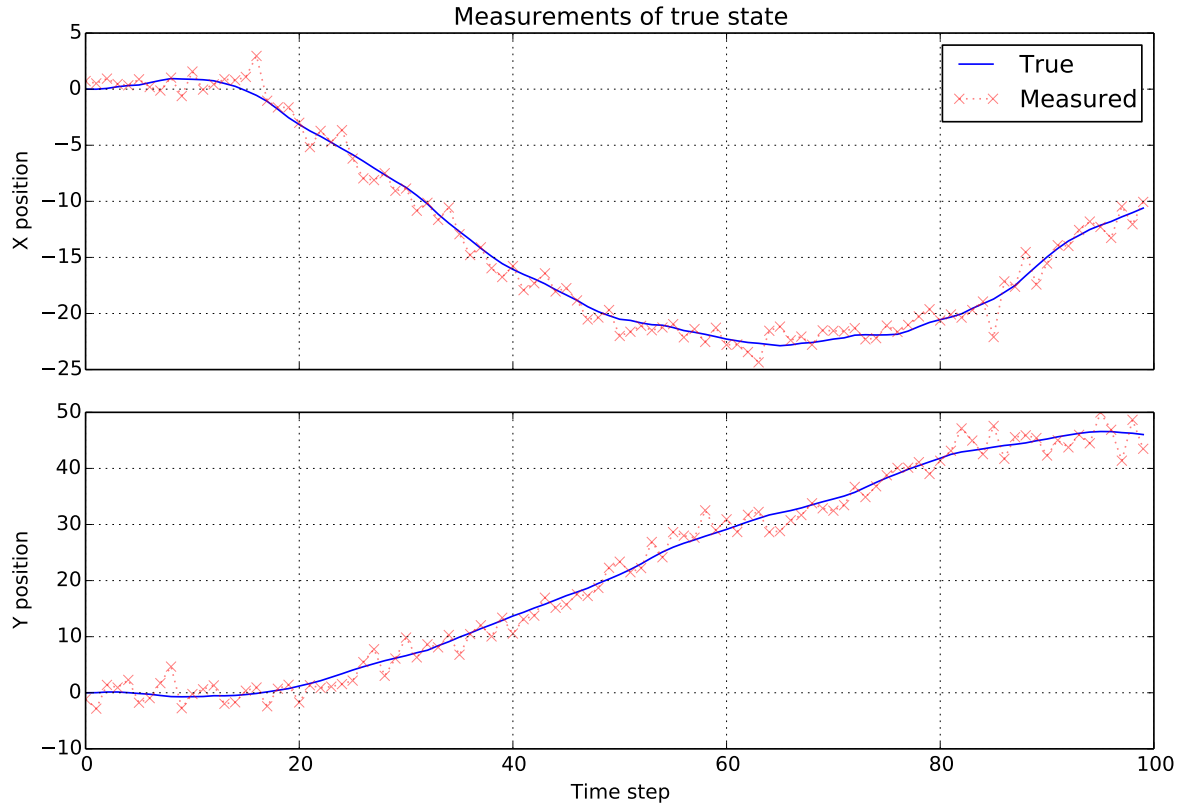
We will use a measurement model where the velocity is a "hidden" state and we can only directly measure position. We'll also specify a measurement error covariance.

```
# We only measure position
H = array([
    [1, 0, 0, 0],
    [0, 1, 0, 0],
])

# And we measure with some error. Note that we have difference
# variances for x and y.
R = diag([1.0, 2.0]) ** 2

# Specify the measurement vector length
MEAS_DIM = H.shape[0]
```

From the measurement matrix and measurement error we can generate noisy measurements from the true states.

```
# Measure the states
from starman.linearsystem import measure_states
measurements = measure_states(true_states, measurement_matrix=H,
                              measurement_covariance=R)
```

Let's plot the measurements overlaid on the true states.



Using the Kalman filter

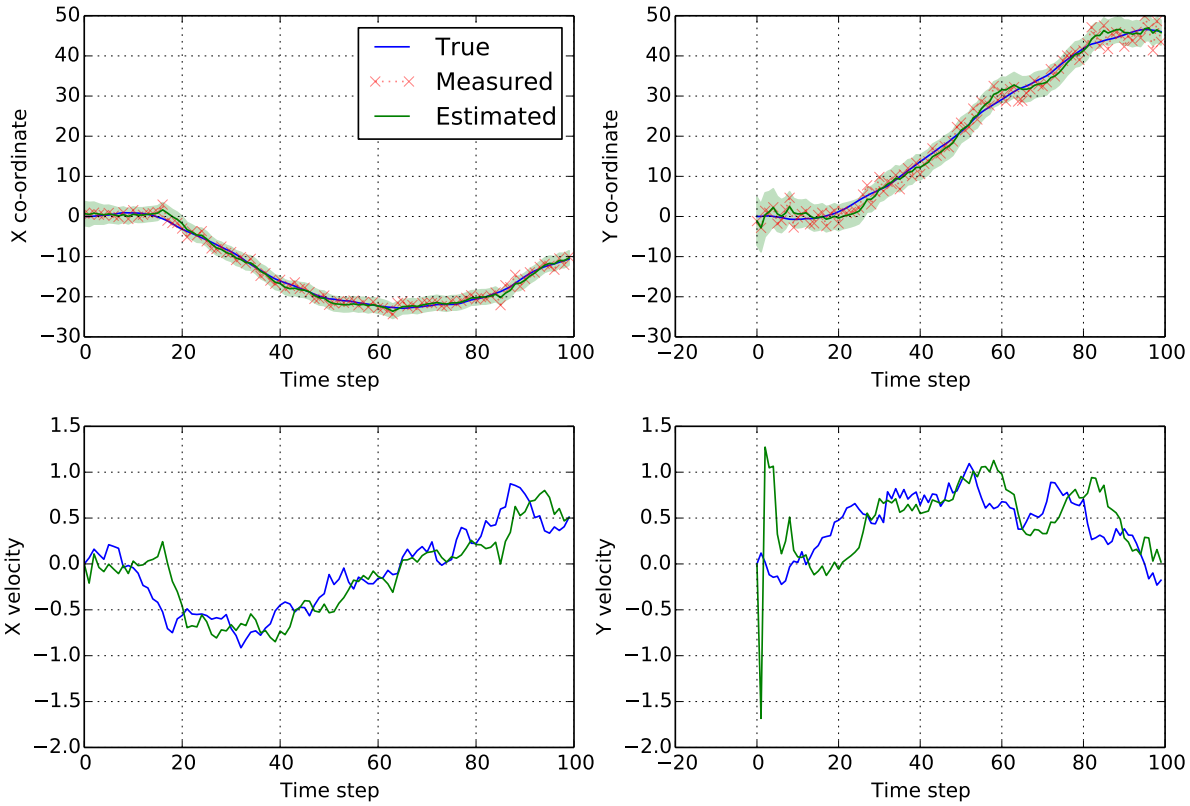
We can create an instance of the `starman.KalmanFilter` to filter our noisy measurements.

```
from starman import KalmanFilter, MultivariateNormal

# Create a kalman filter with constant process matrix and covariances.
kf = KalmanFilter(state_length=STATE_DIM,
                  process_matrix=F, process_covariance=Q)

# For each time step
for k, z in enumerate(measurements):
    # Predict state for this timestep
    kf.predict()

    # Update filter with measurement
    kf.update(measurement=MultivariateNormal(mean=z, cov=R),
              measurement_matrix=H)
```

The *starman.KalmanFilter* class has a number of attributes which give useful information on the filter:

```
# Check that filter length is as expected
assert kf.state_count == N

# Check that the filter state dimension is as expected
assert kf.state_length == STATE_DIM
```

Now we've run the filter, we can see how it has performed. We also shade the three sigma regions for the estimates.



We see that the estimates of position and velocity improve over time.

2.1.3 Rauch-Tung-Striebel smoothing

The Rauch-Tung-Striebel (RTS) smoother provides a method of computing the "all data" *a posteriori* estimate of states (as opposed to the "all previous data" estimate). Assuming there are n time points

in the filter, then the RTS computes the *a posteriori* state estimate at time k after all the data for n time steps are known, $\hat{x}_{k|n}$, and corresponding covariance, $P_{k|n}$, recursively:

$$\hat{x}_{k|n} = \hat{x}_{k|k} + C_k(\hat{x}_{k+1|n} - \hat{x}_{k+1|k}), \quad P_{k|n} = P_{k|k} + C_k(P_{k+1|n} - P_{k+1|k})C_k^T$$

with $C_k = P_{k|k}F_{k+1}^T P_{k+1|k}^{-1}$.

The RTS smoother is an example of an "offline" algorithm in that the estimated state for time step k depends on having seen *all* of the measurements rather than just the measurements up until time k.

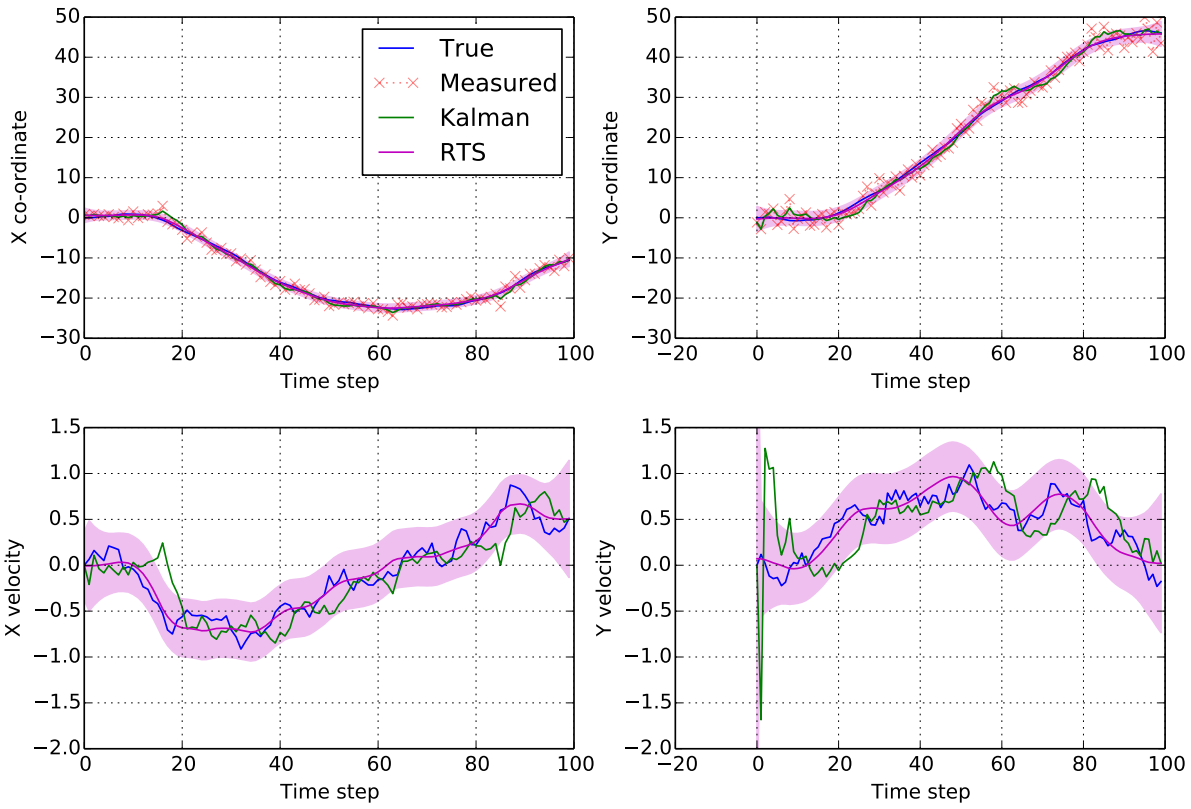
Using RTS smoothing

We'll start by assuming that the steps in *Example: the constant velocity model* have been performed. Namely that we have some true states in `true_states`, measurements in `measurements` and a `starman.KalmanFilter` instance in `kf`.

Following on from that example, we can use the `starman.rts_smooth()` function to compute the smoothed state estimates given all of the data.

```
from starman import rts_smooth

# Compute the smoothed states given all of the data
rts_estimates = rts_smooth(kf)
```

Again, we can plot the estimates and shade the three sigma region.



We can see how the RTS smoothed states are far smoother than the forward estimated states. But that the true state values are still very likely to be within our three sigma band.

2.1.4 Mathematical overview

The Kalman filter alternates between a *predict* step for each time step and zero or more *update* steps. The predict step forms an *a priori* estimate of the state given the dynamics of the system and the update

step refines an *a posteriori* estimate given the measurement.

A Priori Prediction

At time k we are given a state transition matrix, F_k, and estimate of the *process noise*, Q_k. Our *a priori* estimates are then given by:

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k, \quad P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k.$$

Innovation

At time k we are given a matrix, H_k, which specifies how a given measurement is derived from the state and some estimate of the measurement noise covariance, R_k. We may now compute the innovation, y_k, of the measurement from the predicted measurement and our expected innovation covariance, S_k:

$$y_k = z_k - H_k \hat{x}_{k|k-1}, \quad S_k = H_k P_{k|k-1} H_k^T + R_k.$$

Update

We now update the state estimate with the measurement via the so-called *Kalman gain*, K_k:

$$K_k = P_{k|k-1} H_k^T S_k^{-1}.$$

Merging is straightforward. Note that if we have no measurement, our *a posteriori* estimate reduces to the *a priori* one:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k y_k, \quad P_{k|k} = P_{k|k-1} - K_k H_k P_{k|k-1}.$$

CHAPTER 3

Feature Association

When estimating the state of a single system, techniques such as Kalman filtering can be extremely useful. Real situations often have several systems acting independently each of which can generate a measurement. Sometimes it is clear which measurement has arisen from which system. Sometimes it is not. *Feature association* is the process of associating actual measurements with predicted measurements from a set of tracked systems.

3.1 Scott and Longuet-Higgins association

The Scott and Longuet-Higgins algorithm *[SLH]* is an elegant algorithm for associating two sets of features by considering the Gaussian weighted distances between each pair of features. Since it considers pair-wise distances, and then uses an SVD, its computational complexity is approximately $O(n^2)$.

In tracking problems we can use the SLH algorithm when we have good estimates of the predicted measurement estimate covariance and actual measurement covariance.

3.1.1 Example: 2D tracking

Let's first of all create a list of "true" 2D locations of some set of targets. We'll simply sample their location uniformly from the interval $(0, 10] \times (0, 10]$:

```
# Import plotting and numpy functions
from matplotlib.pylab import *

# How many targets?
n_targets = 25

# Sample the ground truth (gt) positions
gt_positions = 10 * np.random.rand(n_targets, 2)
```

We will simulate some tracking problem by assuming we've been tracking the targets and creating some estimates of state. We'll let a certain proportion of the ground truth states be "new" states.

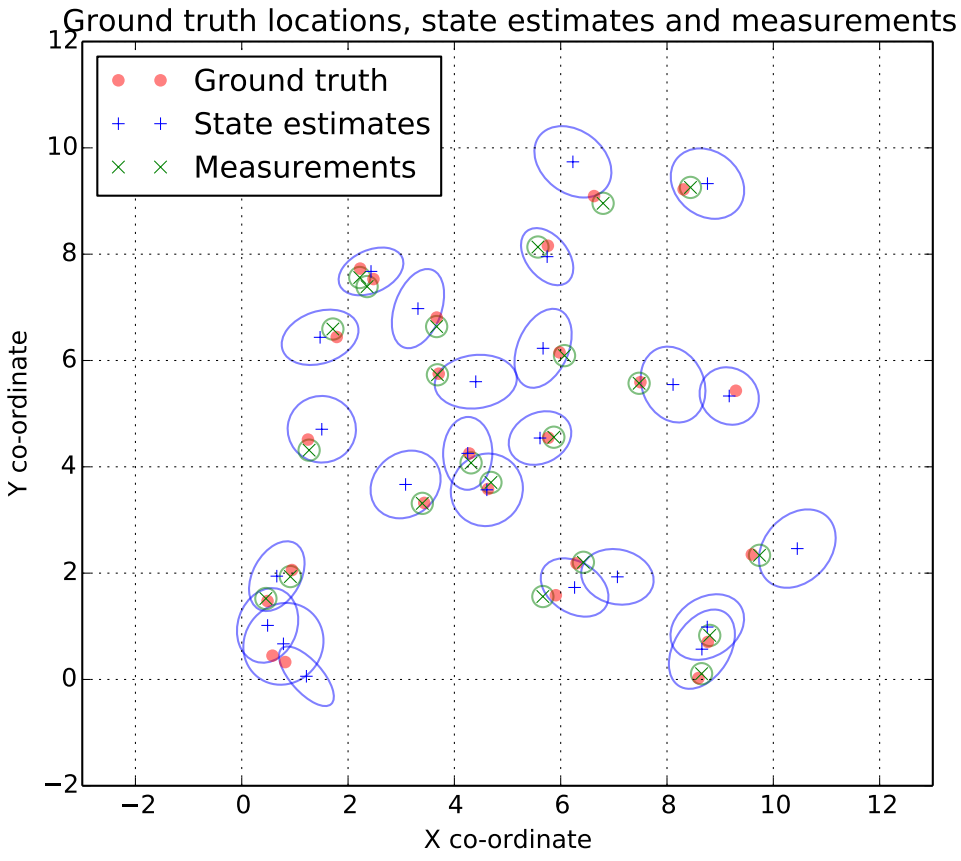
```
from numpy.random import multivariate_normal as sample_mvn
from starman import MultivariateNormal

def simulate_tracking_state(ground_truth):
    """Given a ground truth location, return a MultivariateNormal
    representing a simulated tracking state."""

    # Sample estimate covariance.
    cov = 1e-1 * np.diag(5e-1 + np.random.rand(2))
    cov[1, 0] = cov[0, 1] = 1e-1 * (np.random.rand() - 0.5)

    # Sample mean of estimate based on covariance
```

```
    mean = sample_mvn(mean=ground_truth, cov=cov)

    return MultivariateNormal(mean, cov)

# Sample our simulated state estimates. There's a probability of 0.1 that
# the ground truth state is a new one for this time step.
estimates = [simulate_tracking_state(s)
             for s in gt_positions if np.random.rand() < 0.9]
```

Now we'll simulate measurements on the ground truth states. Again there is a proportion of states which we do not measure but each measurement has the same covariance.

```
from starman.linearsystem import measure_states

# Set measurement covariance
measurement_covariance = np.diag([1e-1, 1e-1]) ** 2

# Get list of MultivariateNormal instances for each measurement. We have
# a probability of 0.1 of missing a state.
gt_measurements = measure_states(gt_positions, np.eye(2),
                                 measurement_covariance)
measurements = [
    MultivariateNormal(mean=measurement, cov=measurement_covariance)
    for measurement in gt_measurements if np.random.rand() < 0.9
]
```

Let's take a look at out ground truth positions and current tracking state estimates. We'll plot a 2-sigma ellipse around each state estimate and each measurement.



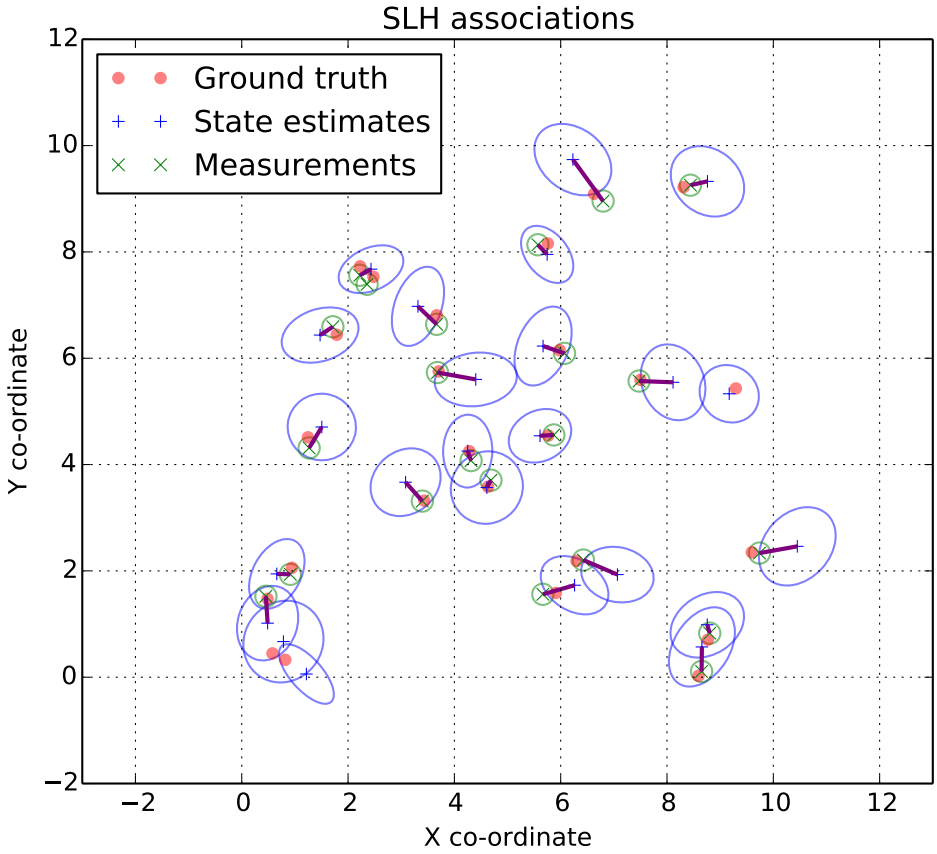
The SLH algorithm is implemented in the *slh_associate()* function. It takes as non-optional arguments two lists of *MultivariateNormal* instances which should be associated. It also takes an optional parameter giving the maximum number of standard deviations two features can be separated

before they are considered to be impossible to associate. In this example we'll use the default 5-sigma separation threshold.

```
from starman import slh_associate

# Use slh_associate to associate state estimates with measurements.
associations = slh_associate(estimates, measurements)

# Associations are represented by an Nx2 array of indices into the two
# lists.
assert associations[:, 0].max() < len(estimates)
assert associations[:, 1].max() < len(measurements)
```

The associations are returned as a two-column array. The first column contains indices into the first list of features and the second column contains indices into the second list. For example we could turn the associations into a list of state estimate mean, measurement pairs:

```
associated_positions = []
for est_idx, meas_idx in associations:
    associated_positions.append([
        estimates[est_idx].mean, measurements[meas_idx].mean
    ])
```



3.1.2 Mathematical overview

The SLH algorithm starts by assuming that there are two sets of features. Each feature is parametrised by a mean and covariance. We shall notate the i-th mean of group k as $\mu_i^{(k)}$ and the i-th covariance of

group k as $\Sigma_i^{(k)}$. We then form a Gaussian weighted proximity matrix, G, where

$$G_{ij} = \exp\left(-\frac{1}{2}\left(\mu_i^{(1)} - \mu_j^{(2)}\right)^T \left(\Sigma_i^{(1)} + \Sigma_j^{(2)}\right)^{-1} \left(\mu_i^{(1)} - \mu_j^{(2)}\right)\right).$$

Our intution is that "true" associations are represented by a) a value close to 1 in G and b) that value being the largest in both its row and column. The "ideal" G is one where there is at most a single 1 in each row an column and every other element is zero. (This ideal matrix being orthogonal.) The SLH algorithm attempts to magnify the orthogonality of G by way of the singular value decomposition (SVD).

One firstly takes the SVD of G which finds U, S and V such that

$$U\,S\,V^T = G.$$

The matrix of singular values S only has non-zero elements on its diagonal. Form a new matrix Λ from S by setting all non-zero elements to 1. Then form P as

$$P = U\,\Lambda\,V^T.$$

Associate feature i in list 1 with feature j in list 2 if and only if:

1. Element P_{ij} is the maximum in its row and column.
2. G_{ij} is greater than some association threshold, α.

In practice the association threshold is set with reference to some number of standard deviations, σ. So, $\alpha = \exp(-\sigma^2/2)$.

The SLH algorithm can be interpreted as minimising the sum of squared distances between features where those distances are normalised by the covariance matrices of the features.

CHAPTER 4

Programmer's Reference

Below is a description of the public API of starman separated by functionality. In-depth discussion how to use the API can be found in the appropriate sections of the documentation.

4.1 State estimation

class starman.**KalmanFilter**(*initial_state_estimate=None*, *process_matrix=None*, *process_covariance=None*, *control_matrix=None*, *state_length=None*)

A KalmanFilter maintains an estimate of true state given noisy measurements.

The filter is initialised to have no state estimates. (Time step "-1" if you will.) Before calling *update()*, *predict()* must be called at least once.

The filter represents its state estimates as frozen *MultivariateNormal* instances.

Parameters

- **initial_state_estimate** (*None or MultivariateNormal*) – The initial estimate of the true state used for the first *predict()* step. If *None*, *state_length* must be specified and the initial state estimate is initialised to zero mean and a covariance of the identity matrix muiltiplied by a large value. (Specifically the value of KalmanFilter.LARGE_COVARIANCE.)
- **process_matrix** (*array or None*) – The process matrix to use if none is passed to *predict()*.
- **process_covariance** (*array or None*) – The process noise covariance to use if none is passed to *predict()*.
- **control_matrix** – (array or None): The control matrix to use if none is passed to *predict()*.
- **state_length** (*None or int*) – Must only be specified if *initial_state_estimate* is None. In which case, this is used as the length of the state vector.

Raises ValueError – The passed matrices have inconsistent or invalid shapes.

prior_state_estimates
list of MultivariateNormal

Element *k* is the the *a priori* state estimate for time step *k*.

posterior_state_estimates
list of MultivariateNormal

Element *k* is the the *a posteriori* state estimate for time step *k*.

measurements
list of list of MultivariateNormal

Element *k* is a list of `MultivariateNormal` instances. These are the instances passed to `update()` for time step *k*.

process_matrices
list of array

Element *k* is the process matrix used by `predict()` at time step *k*.

process_covariances
list of array

Element *k* is the process covariance used by `predict()` at time step *k*.

measurement_matrices
list of list of array

Element *k* is a list of the measurement matrices passed to each call to `update()` for that time step.

state_length
int

Number of elements in the state vector.

clone()
Return a new `KalmanFilter` instance which is a shallow clone of this one. By "shallow", although the lists of measurements, etc, are cloned, the `MultivariateNormal` instances within them are not. Since `predict()` and `update()` do not modify the elements of these lists, it is safe to run two cloned filters in parallel as long as one does not directly modify the states.

Returns *(KalmanFilter)* – A new `KalmanFilter` instance.

measurement_count
Property returning the total number of measurements which have been passed to this filter.

predict(*control=None*, *control_matrix=None*, *process_matrix=None*, *process_covariance=None*)
Predict the next *a priori* state mean and covariance given the last posterior. As a special case the first call to this method will initialise the posterior and prior estimates from the *initial_state_estimate* and *initial_covariance* arguments passed when this object was created. In this case the *process_matrix* and *process_covariance* arguments are unused but are still recorded in the `process_matrices` and `process_covariances` attributes.

Parameters

- **control** (*array or None*) – If specified, the control input for this predict step.
- **control_matrix** (*array or None*) – If specified, the control matrix to use for this time step.
- **process_matrix** (*array or None*) – If specified, the process matrix to use for this time step.
- **process_covariance** (*array or None*) – If specified, the process covariance to use for this time step.

state_count
Property returning the number of states/time steps this filter has processed. Since the first time step is always 0, the final index will always be `state_count` - 1.

truncate(*new_count*)
Truncate the filter as if only *new_count* `predict()`, `update()` steps had been performed. If *new_count* is greater than `state_count` then this function is a no-op.

Measurements, state estimates, process matrices and process noises which are truncated are discarded.

Parameters **new_count** (*int*) – Number of states to retain.

update(*measurement*, *measurement_matrix*)
After each `predict()`, this method may be called repeatedly to provide additional measurements for each time step.

Parameters

- **measurement** (`MultivariateNormal`) – Measurement for this time step with specified mean and covariance.
- **measurement_matrix** (*array*) – Measurement matrix for this measurement.

starman.**rts_smooth**(*kalman_filter*, *state_count=None*)
Compute the Rauch-Tung-Striebel smoothed state estimates and estimate covariances for a Kalman filter.

Parameters

- **kalman_filter** (`KalmanFilter`) – Filter whose smoothed states should be returned
- **state_count** (*int or None*) – Number of smoothed states to return. If None, use `kalman_filter.state_count`.

Returns *(list of MultivariateNormal)* – List of multivariate normal distributions. The mean of the distribution is the estimated state and the covariance is the covariance of the estimate.

4.2 Feature association

starman.**slh_associate**(*a_features*, *b_features*, *max_sigma=5*)
An implementation of the Scott and Longuet-Higgins algorithm for feature association.

This function takes two lists of features. Each feature is a `MultivariateNormal` instance representing a feature location and its associated uncertainty.

Parameters

- **a_features** (*list of MultivariateNormal*) –
- **b_features** (*list of MultivariateNormal*) –
- **max_sigma** (*float or int*) – maximum number of standard deviations two features can be separated and still considered "associated".

Returns *(array)* – A Nx2 array of feature associations. Column 0 is the index into the a_features list, column 1 is the index into the b_features list.

4.3 Representation of state estimates

class starman.**MultivariateNormal**(*mean=None*, *cov=None*)
MultivariateNormal represents a multivariate normal (or "Gaussian") distribution parametrised in terms of a mean and covariance. The mean is a length-N vector and the covariance is a NxN matrix.

If mean is unspecified, it defaults to a zero-filled vector whose dimension matches the covariance.

If the covariance is unspecified, it defaults to an identity matrix whose shape matches the dimension of the mean.

If neither mean or covariance are specified, default values of 0 and 1 are used.

Parameters

- **mean** (*None or array*) – Distribution mean.
- **cov** (*None or array*) – Distribution covariance.

rvs(*size=1*)

Convenience method to sample from this distribution.

Parameters **size** (*int or tuple*) – Shape of return value. Each element is drawn independently from this distribution.

4.4 Helper functions for linear systems

The *starman.linearsystem* module contains some helper functions for systems with linear dynamics and a linear measurement model.

starman.linearsystem.**generate_states**(*state_count*, *process_matrix*, *process_covariance*, *initial_state=None*)

Generate states by simulating a linear system with constant process matrix and process noise covariance.

Parameters

- **state_count** (*int*) – Number of states to generate.
- **process_matrix** (*array*) – Square array
- **process_covariance** (*array*) – Square array specifying process noise covariance.
- **initial_state** (*array or None*) – If omitted, use zero-filled vector as initial state.

starman.linearsystem.**measure_states**(*states*, *measurement_matrix*, *measurement_covariance*)

Measure a list of states with a measurement matrix in the presence of measurement noise.

Parameters

- **states** (*array*) – states to measure. Shape is NxSTATE_DIM.
- **measurement_matrix** (*array*) – Each state in *states* is measured with this matrix. Should be MEAS_DIMxSTATE_DIM in shape.
- **measurement_covariance** (*array*) – Measurement noise covariance. Should be MEAS_DIMxMEAS_DIM.

Returns *(array)* – NxMEAS_DIM array of measurements.

[SLH] Scott, Guy L., and H. Christopher Longuet-Higgins. "An algorithm for associating the features of two images." Proceedings of the Royal Society of London B: Biological Sciences 244.1309 (1991): 21-26.

S

C

G

K

M

P

R

S

T

U