

### 📌 Note

Click [here](#) to download the full example code or to run this example in your browser via Binder

## 12 - An introduction to Stone Soup: using the Information filter

This notebook is designed to introduce the Information Filter using a single target scenario as an example.

### Background and notation:

The information filter can be used when there is large, or infinite uncertainty about the initial state of the object. To compute the predicting and updating steps, the infinite covariance is therefore converted to its inverse, using both the Information matrix and the Information state.

- We begin by creating a constant velocity model with  $q = 0.05$
- A 'truth path' is created starting at (20,20) moving to the NE at one distance unit per (time) step

in each dimension. \* We propagate this with the transition model to generate a ground truth path

Firstly, we run the general imports, set the start time and build the Ground Truth constant velocity model. This follows the same procedure as that in Tutorial 1 - Kalman Filter.

```
import numpy as np

from datetime import datetime, timedelta
start_time = datetime.now()

np.random.seed(1991)

# setting up ground truth

from stonesoup.types.groundtruth import GroundTruthPath, GroundTruthState
from stonesoup.models.transition.linear import (CombinedLinearGaussianTransitionModel,
                                                ConstantVelocity)

transition_model =
CombinedLinearGaussianTransitionModel([ConstantVelocity(0.05), ConstantVelocity(0.05)])

# Creating the initial truth state.

truth = GroundTruthPath([GroundTruthState([20, 1, 20, 1], timestamp=start_time)])

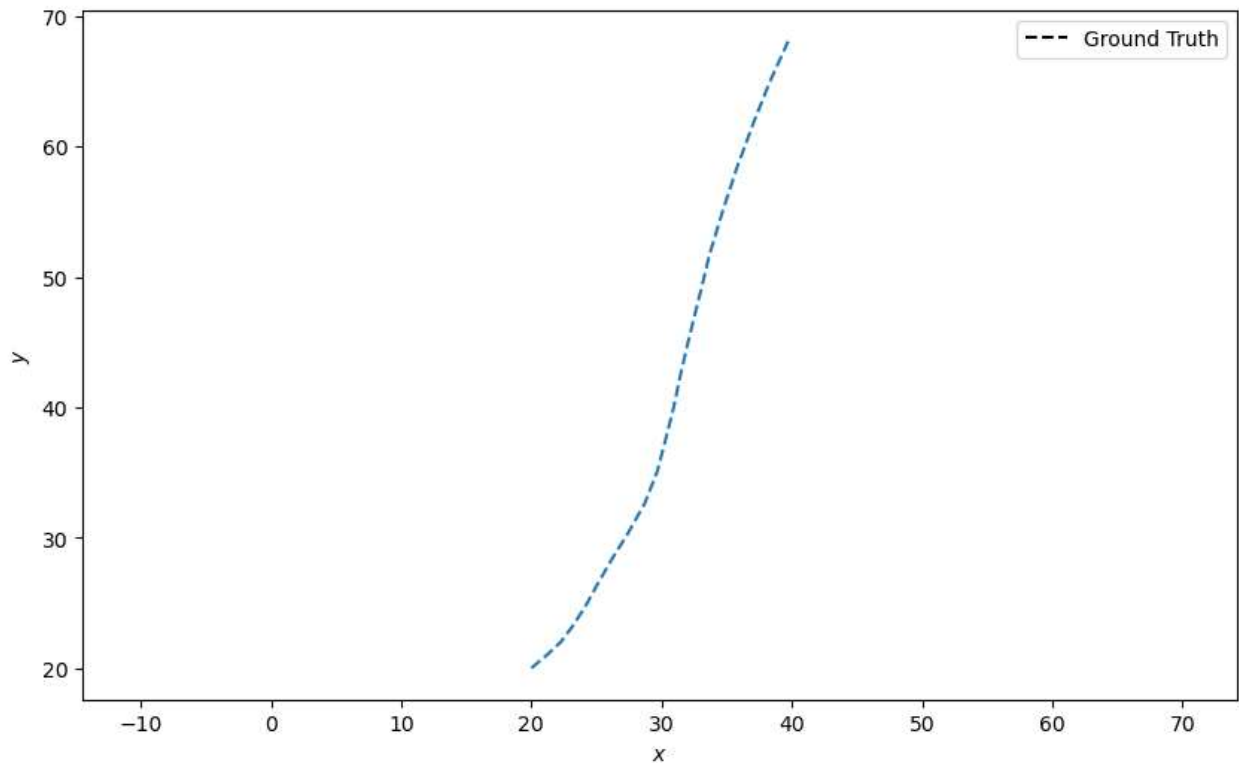
# Generating the Ground truth path using a transition model.

for k in range(1, 21):
    truth.append(GroundTruthState(
        transition_model.function(truth[k-1], noise=True, time_interval=timedelta(seconds=1)),
        timestamp=start_time + timedelta(seconds=k)))
```

Importing the `Plotter` class from Stone Soup we can plot the results. Note that the mapping argument is `[0, 2]` because those are the  $x$  and  $y$  position indices from our state vector.

## Plotting:

```
from stonesoup.plotter import Plotter
plotter = Plotter()
plotter.plot_ground_truths(truth, [0, 2])
```



## Taking Measurements:

As per the original Kalman tutorial, we'll use one of Stone Soup's measurement models in order to generate measurements from the ground truth. We shall assume a 'linear' sensor which detects the position only (not the velocity) of a target.

Omega is set to 5.

The linear Gaussian measurement model is set up by indicating the number of dimensions in the state vector and the dimensions that are measured (specifying  $H_k$ ) and the noise covariance matrix  $R$ .

```
# measurements
from stonesoup.types.detection import Detection
from stonesoup.models.measurement.linear import LinearGaussian

measurement_model = LinearGaussian(ndim_state = 4, # Number of state dimensions (position and
velocity in 2D)
                                   mapping=(0,2), # Mapping measurement vector index to state index
                                   noise_covar=np.array([[5,0], # Covariance matrix for Gaussian PDF
                                                         [0,5]]))

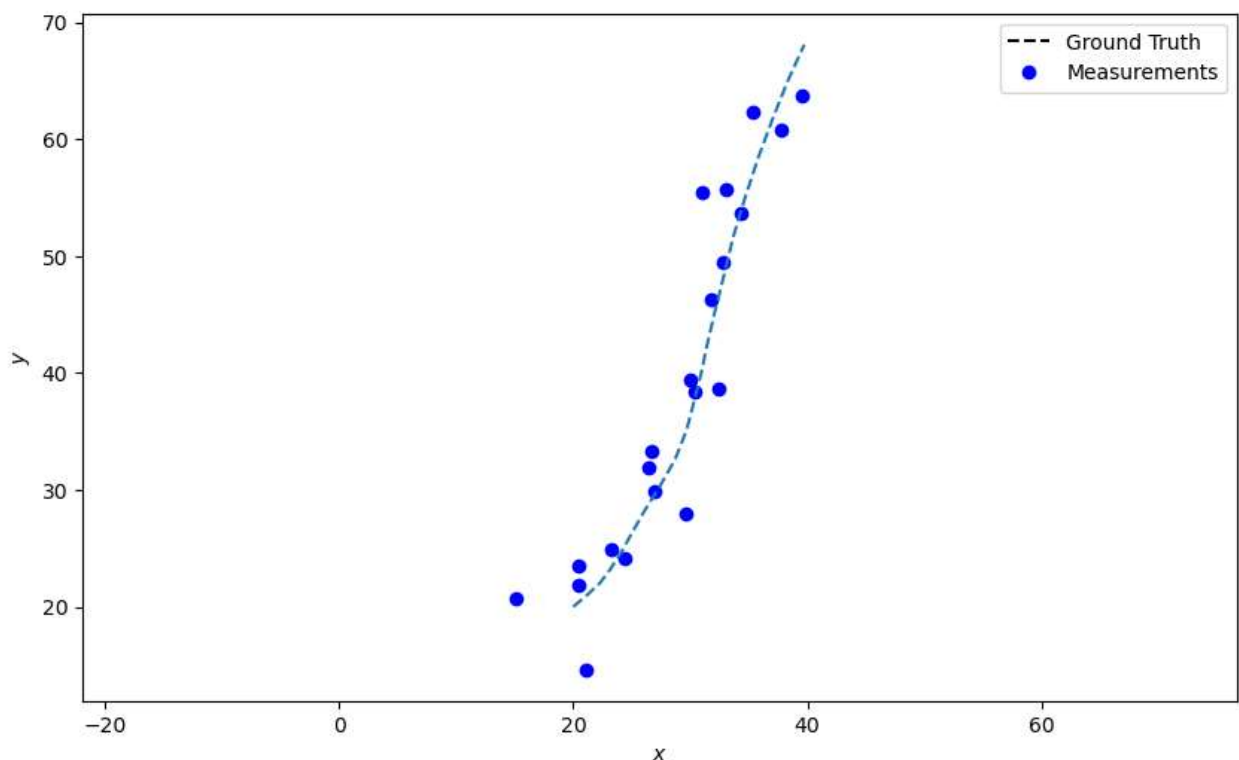
# taking measurements

measurements = []

for state in truth:
    measurement = measurement_model.function(state,noise=True)
    measurements.append(Detection(measurement,timestamp=state.timestamp,measurement_model =
measurement_model))
```

We plot the measurements using the Plotter class in Stone Soup. Again specifying the  $x$ ,  $y$  position indices from the state vector.

```
plotter.plot_measurements(measurements,[0,2])
plotter.fig
```



## Running the Information Filter:

We must first import the `InformationKalmanPredictor`, and `InformationKalmanUpdater` from the corresponding libraries.

As before, the Predictor must be passed a Transition model, and the updater a Measurement model.

It is important to note that, unlike the Kalman Filter in Tutorial 1, the Information Filter requires the prior estimate to be in the form of an `InformationState` (not a `GaussianState`).

The `InformationState` can be imported from `stonesoup.types.state`, and takes arguments: Information state, Information Matrix and a timestamp.

The information matrix is defined as :  $Y_{k-1} = [P_{k-1}]^{-1}$ . That is, the inverse of the covariance matrix. The information state is defined as :  $y_{k-1} = [P_{k-1}]^{-1} x_{k-1}$ .

That is the matrix multiplication of the information matrix and the prior state in  $x_{k-1}$ .

Using the same prior state as the original Kalman filtering example, we must firstly convert the covariance to be the Information matrix, and calculate the `InformationState`,  $y_{k-1}$ .

```
# predicting and updating
from stonesoup.predictor.information import InformationKalmanPredictor
from stonesoup.updater.information import InformationKalmanUpdater

# Creating Information predictor and updater objects
predictor = InformationKalmanPredictor(transition_model)
updater = InformationKalmanUpdater(measurement_model)
```

As before, we use the `SingleHypothesis` class. The explicitly associates a single predicted state to a single detection.

```

from stonesoup.types.state import InformationState
from stonesoup.types.state import GaussianState

# Precision matrix - the inverse of the covariance matrix
kalman_covar = np.diag([1.5, 0.5, 1.5, 0.5])
inverse_kal_covar = np.linalg.inv(kalman_covar) # information matrix

#  $y_{k-1} = P_{k-1}^{-1} @ x_{k-1}$ 
kalman_state = [20, 1, 20, 1]
information_state = kalman_state @ inverse_kal_covar

# Must use information state with precision matrix instead of Gaussian state
prior = InformationState(information_state, inverse_kal_covar, timestamp=start_time)

from stonesoup.types.hypothesis import SingleHypothesis
from stonesoup.types.track import Track

track = Track()
track2 = Track()

for measurement in measurements:
    prediction = predictor.predict(prior, timestamp=measurement.timestamp)
    hypothesis = SingleHypothesis(prediction, measurement)
    post = updater.update(hypothesis)
    track.append(post)
    prior = track[-1]

# Converting InformationState to GaussianState
# Note: You can use property .gaussian_state to convert the information state and information matrix.
for state in track:
    covariance = np.linalg.inv(state.precision)
    kalman_state = covariance @ state.state_vector
    track2.append(GaussianState(kalman_state, covariance, timestamp = state.timestamp))

```

## Plotting:

Because the results are given as InformationState data types, we must convert these back to GaussianState types for the Plotting library to work correctly.

The below illustrates what happens if we do not convert back.

To work around this, we therefore recalculate the covariance of each state in the track to take its inverse (giving the gaussian covariance). We also calculate the kalman\_state equivalent by multiplying by the covariance.

$$y_{k-1} = [P_{k-1}]^{-1} x_{k-1}.$$

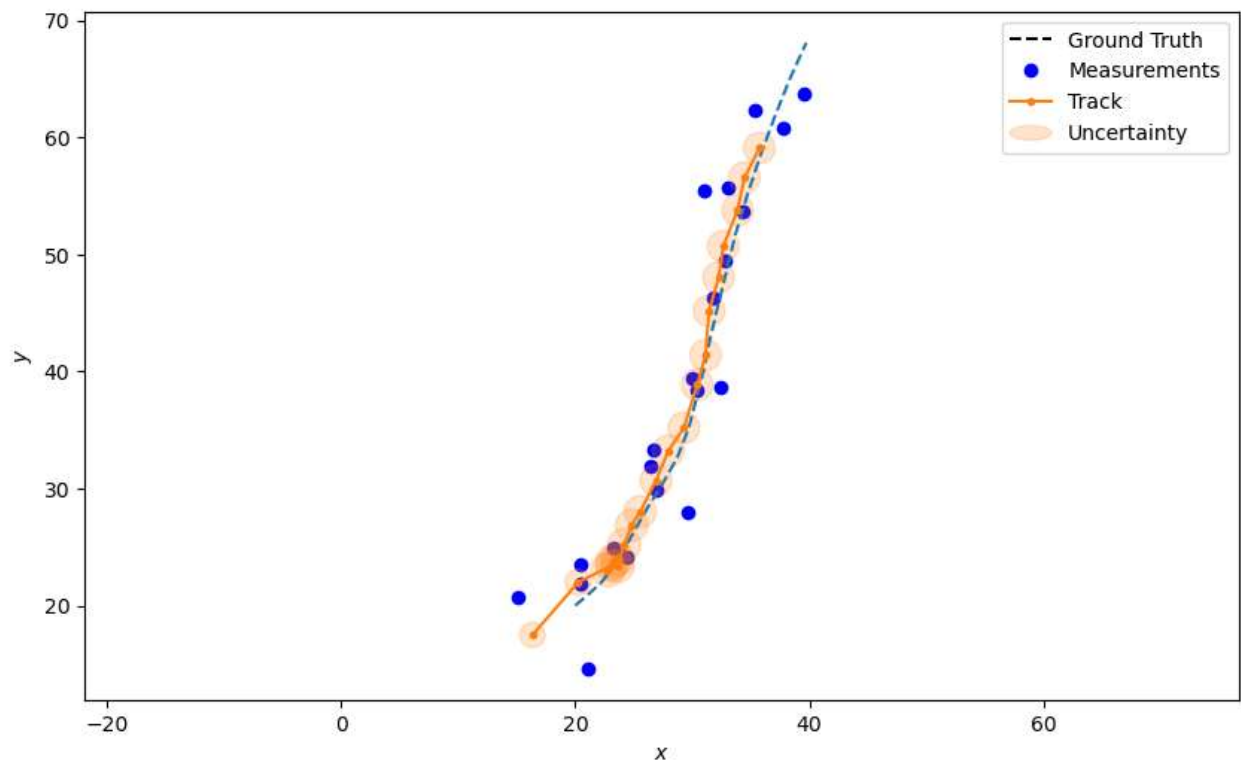
In order to derive  $x_{k-1}$ , we multiply by the covariance matrix.

Appending these to Track2 as a GaussianState data type, we are able to plot correctly (see second figure).

```

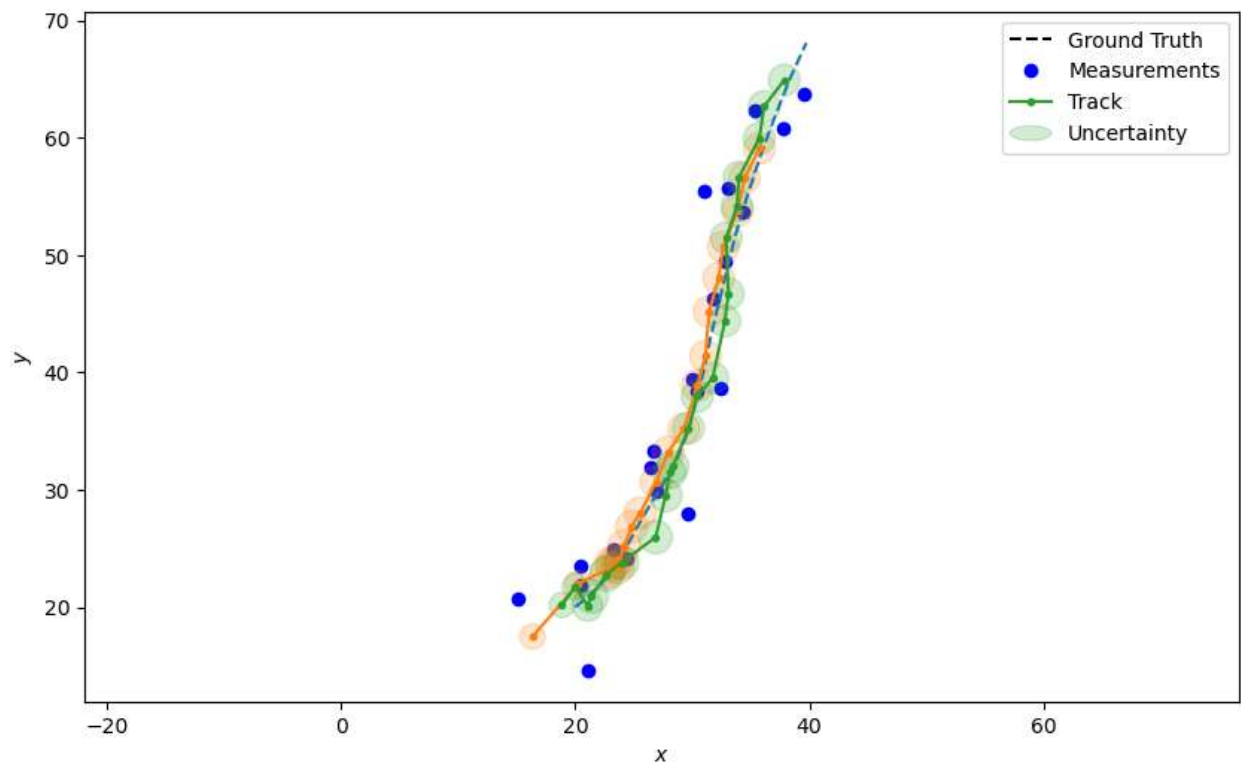
plotter.plot_tracks(track, [0, 2], uncertainty=True)
plotter.fig

```





Corrected Plot using Gaussian State:

```
plotter.plot_tracks(track2,[0,2],uncertainty=True)
plotter.fig
```



Total running time of the script: ( 0 minutes 2.778 seconds)

 Download Python source code: `12_InformationFilterTutorial.py`

 Download Jupyter notebook: `12_InformationFilterTutorial.ipynb`