Deception - Hiding Example:

The following example shows the implementation of Stone Soup against an object exhibiting deceptive behaviour (hiding).

We use a Constant Jerk Model to specify waypoints that the platform will pass through. This produces 2 ground truth paths beginning and ending in the same positions. One proceeds 'around a bend' (e.g. building, object etc.) and undergoes deception. Within this period, we assume that the sensor will not be able to make any detections.

In [1]:
```
from datetime import datetime
from datetime import timedelta
from matplotlib import pyplot as plt
from ordered_set import OrderedSet

import numpy as np

# Stone Soup imports:
from stonesoup.types.state import State, GaussianState
from stonesoup.types.array import StateVector
from stonesoup.types.array import CovarianceMatrix
from stonesoup.models.transition.linear import (
    CombinedLinearGaussianTransitionModel, ConstantVelocity)
```

Creating the Ground Truth Paths:

Begin by specifying the waypoints for each of the paths.

We then loop through the state list and pass the StateVectors (as well as the position and velocity mappings) to the ConstantJerkSimulator's **'create_models()'** method. This produces transition models and transition times which are then used to propagate the platform across the time period, moving between waypoints.

The **MultiTransitionMovingPlatform** is uses the output transition models and transition times. We move the platform every second, and add this to the platform set.

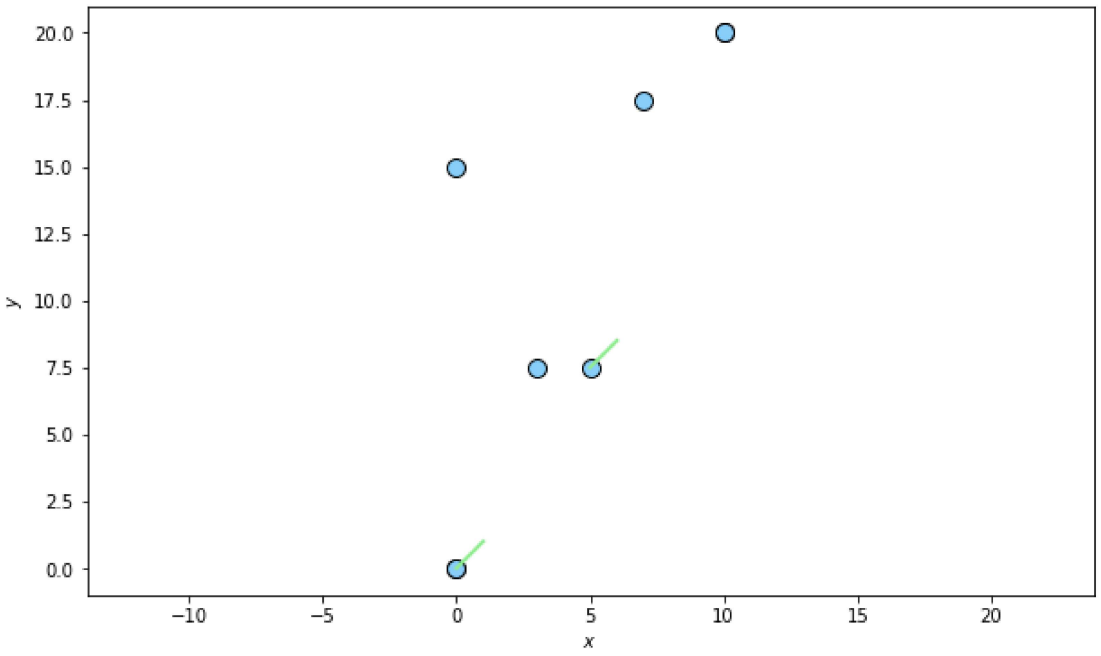
In [2]:
```
# Defining start positions
from stonesoup.platform import MultiTransitionMovingPlatform
from stonesoup.simulator.transition import ConstantJerkSimulator

start = datetime.now()
position_mapping = (0, 2)
velocity_mapping = (1, 3)

state_list = []

states1 = [
    State(StateVector([0, 1, 0, 1]), start),  # at origin, pointing NE
    State(StateVector([3,1,7.5,1]),start+timedelta(seconds=5)),
    State(StateVector([0, 1, 15, 1]), start + timedelta(seconds=10)),  # at (0, 15), keep same
    State(StateVector([7, 1, 17.5, 1]), start + timedelta(seconds=15)),
    State(StateVector([10, 0, 20, 0]), start + timedelta(seconds=20))  # at (10, 20), 0 velocit
]

states2 = [
    State(StateVector([0, 1, 0, 1]), start),  # at origin, pointing NE
    State(StateVector([5, 1, 7.5, 1]), start + timedelta(seconds=10)),  # at (5, 7.5), keep sam
    State(StateVector([10, 0, 20, 0]), start + timedelta(seconds=20))  # at (10, 20), 0 velocit
]

state_list.append(states1)
state_list.append(states2)
```

```
platform_set = OrderedSet()

for states in state_list:

    # Setting up constant jerk sim

    transition_models, transition_times = ConstantJerkSimulator.create_models(states,
                                                                              position_mapping,
                                                                              velocity_mapping)



    platform = MultiTransitionMovingPlatform(states=states[0],
                                             position_mapping=position_mapping,
                                             transition_models=transition_models,
                                             transition_times=transition_times)

    while platform.timestamp < states[-1].timestamp:
        platform.move(timestamp=platform.timestamp+timedelta(seconds=1))
        platform_set.add(platform)
```

Plotting the platform states to check the route each path will take, and then plotting the ground truth using the Plotter from Stone Soup.

In [3]:

```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.axis('equal')
for states in state_list:
    ax.scatter(*zip(*[state.state_vector[(0, 2), :] for state in states]),
               color='lightskyblue', s=100, edgecolors='black')


for state in states:
    x, vx, y, vy = state.state_vector
    ax.plot((x, x+vx), (y, y+vy), color='lightgreen', linewidth=2)
```

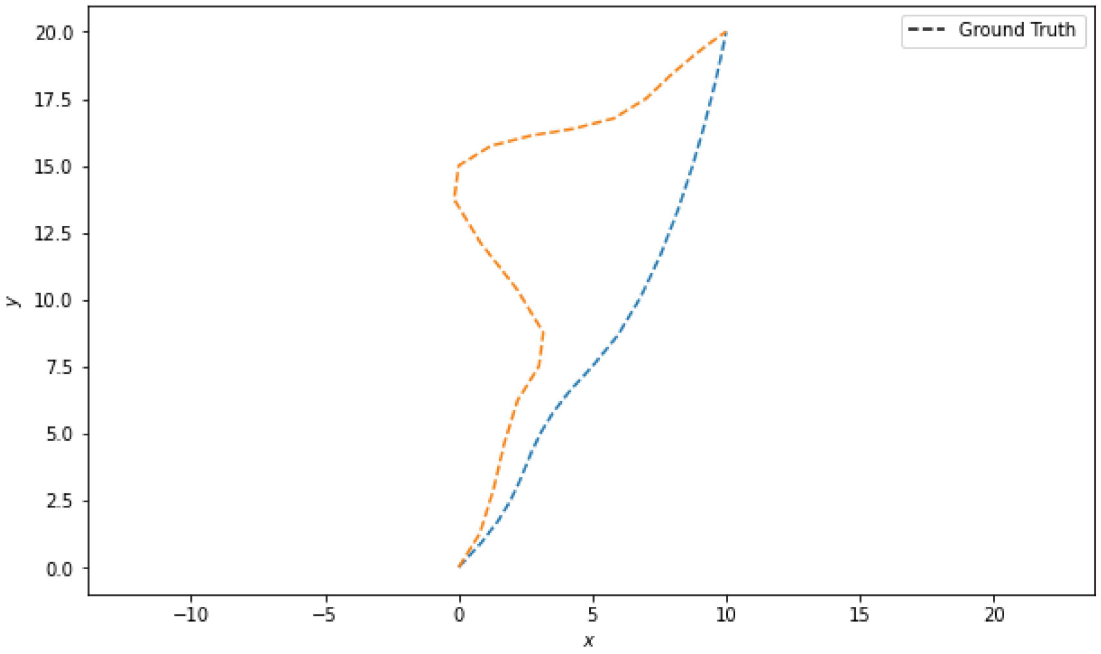


In [ ]:

In [ ]:

In [ ]:

In [4]:
```
from stonesoup.plotter import Plotter

plotter = Plotter()

plotter.plot_ground_truths(set(platform_set),(0,2))
```



In [ ]:
```

```

Taking Measurements:

We use a sensor to take measurements using the **RadarRotatingBearingRange** class. Pass in the position mappings, sensor covariance as well as the state dimensions. We have set the sensor field of view to 360 degrees for this example. The dwell centre includes a StateVector to give the sensor a concept of time (e.g. when each measurement is taken). The second argument defaults to 'start_time'.

We then take the timesteps from the platform states in order to take measurements with the sensor at the correct intervalls.

In [5]:
```
# Taking measurements
# adding a sensor and measurements
from stonesoup.types.state import StateVector
from stonesoup.sensor.radar.radar import RadarRotatingBearingRange
from stonesoup.types.state import State


# Creating a single sensor

sensor = RadarRotatingBearingRange(
    position_mapping=(0, 2),
    noise_covar=np.array([[np.radians(0.5) ** 2, 0],
                          [0, 0.75 ** 2]]),
    ndim_state=4,
    position=np.array([[10], [0]]),
    rpm=60,
    fov_angle=np.radians(360), # Field of view at 360.
    dwell_centre=StateVector([0.0]), # Including State to give the sensor a concept of time. Us
    max_range=np.inf
```

```
)
sensor.timestamp = start
```

In [6]:

```
# Generate list of timesteps from ground truth timestamps
# both truths will have the same timesteps
timesteps = []
for state in platform.states:
    timesteps.append(state.timestamp)
```

Using the sensor

We loop through each of the timesteps, and if we are looping on the first platform (the deceptive path), we then check whether the object has gone around the 'bend' (e.g. y co-ordinate of 10 - 17). If so, we 'continue' and take no measurments.

Otherwise, we check that there is a sufficient probability of detection. If so, we take a measurement. We then append this to the set.

Next, we generate clutter using a uniform distribution around the ground truth x and y values. We add this clutter to the set.

After looping through both platforms, we append the set of measurements to our 'measurements_all' list to ensure we have a list of length equal to the number of timesteps (multiple detections at a single timestep).

In [7]:

```
from stonesoup.types.detection import Clutter
from stonesoup.types.detection import TrueDetection
from scipy.stats import uniform
from stonesoup.models.measurement.linear import LinearGaussian

np.random.seed(101) # set random seed to constant example


measurement_model = LinearGaussian(
    ndim_state=4,
    mapping=(0, 2),
    noise_covar=np.array([[0.75, 0],
                          [0, 0.75]])
    )

prob_detection = 0.9
clutter_set = set()

measurements_all = []
for m, timestep in enumerate(timesteps):

    measurements_sensor = set()

    for n, platform in enumerate(platform_set): # Using OrderedSet to ensure the correct path i
                                                # deception in each case

#         if (n==0) and (platform[timestep].state_vector[2] > 10) and (platform[timestep].state_
#             # If first track / platform, and between y co-ordinate 10 and 17, then no detecti
#             continue

        if (np.random.rand() < prob_detection):
            # If sufficient probability of detection, then take measurement

            if (n==0) and (platform[timestep].state_vector[2] > 10) and (platform[timestep].sta
            # If first track / platform, and between y co-ordinate 10 and 17, then no detection
                None
            else:
                measurements = sensor.measure(OrderedSet([platform[timestep]]),noise=True)
    #             measurements_sensor.add(TrueDetection(state_vector = measurements, timestamp
                measurements_sensor |= measurements #adding measurements to measurements_sensor
```

```
        # Generating Clutter - even when no detections
        truth_x = platform[timestep].state_vector[0]
        truth_y = platform[timestep].state_vector[2]

        for i in range(np.random.randint(2)):
            x = uniform.rvs(truth_x - 4,5)
            y = uniform.rvs(truth_y - 4,5)

            measurements_sensor.add(Clutter(np.array([[x],[y]]),timestamp=timestep,measurement_


    measurements_all.append(measurements_sensor)
```

In [8]:
```
len(measurements_all)
```

Out[8]: 21

In [9]:
```
for platform in OrderedSet(platform_set):
    print(platform[timestep])
```

```
State(
    state_vector=StateVector([[10.],
                              [ 0.],
                              [20.],
                              [ 0.]]),
    timestamp=2022-06-27 16:17:49.552134)
State(
    state_vector=StateVector([[10.],
                              [ 0.],
                              [20.],
                              [ 0.]]),
    timestamp=2022-06-27 16:17:49.552134)
```

In [10]:
```
timestep
```

Out[10]: datetime.datetime(2022, 6, 27, 16, 17, 49, 552134)

In [ ]:
```

```

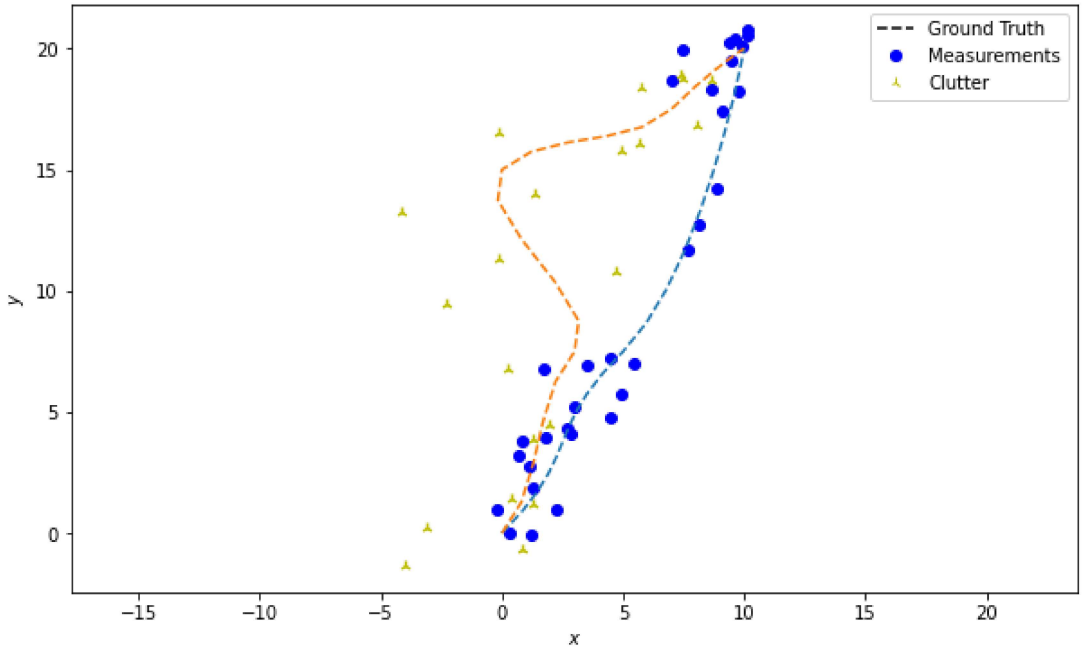
Plotting Measurements

We plot the measurements using the Plotter from Stone Soup. We observe that the path exhibiting deception has no measurements when it moves around the 'bend'. On the other hand, the orange path (no deceptive behaviour) moves from the start to end waypoint and has detections across the entire ground truth.

In [11]:
```
# Plotting measurements
plotter.plot_measurements(measurements_all,[0,2])
```

In [12]:
```
plotter.fig
```

Out[12]:



Using Extended Kalman to predict and update our estimations

We create a transition model (Constant Velocity) to use in the predictor class. The transition models before are generated by our constant jerk model. For our predictions we use a ConstantVelocity model with q = 0.05.

Note: We do not pass a measurement_model to the updater object. This is because the measurement_model is added to the detections by the sensor. The Distance hypothesiser takes the measurement_model from the detection in the .update() method when calculating the 'predicted_measurement'. It will then calcualte the distance between this and the detection itself.

In [13]:
```
# Extended kalman

# creating transition model (constant velocity)

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


# Creating kalman predictor and updater
from stonesoup.predictor.kalman import ExtendedKalmanPredictor
predictor = ExtendedKalmanPredictor(transition_model)

from stonesoup.updater.kalman import ExtendedKalmanUpdater
updater = ExtendedKalmanUpdater(measurement_model=None)
# measurement model is added to detections by the sensor. The Distance hypothesiser below also
# model from the detection when it calcualtes the predicted_measurement. It then calculates the
# and the detection itself.
```

We use the Mahalanobis to measure the distance a point is from a probability distribution. This is then passed into the GlobalNearestNeigbour for our data association problem (2 tracks, clutter).

Initialise 2 prior GaussianState estimates with StateVector and Covariance.

In [14]:
```
from stonesoup.hypothesiser.distance import DistanceHypothesiser
from stonesoup.measures import Mahalanobis
hypothesiser = DistanceHypothesiser(predictor, updater, measure=Mahalanobis(), missed_distance=


from stonesoup.dataassociator.neighbour import GlobalNearestNeighbour
data_associator = GlobalNearestNeighbour(hypothesiser)
```

In [15]:
```
# Creating Prior estimate. Same as start position for the tracks.
from stonesoup.types.state import GaussianState
prior1 = GaussianState([0,1,0,1],np.diag([0.5,0.25,0.5,0.25]),timestamp=start)
prior2 = GaussianState([0,1,0,1],np.diag([0.5,0.25,0.5,0.25]),timestamp=start)
```

Initiator and deleter

We import a deleter and initiator from strone soup and pass in the covariance threshold and the min points. The initiator takes the below arguments.

In [16]:
```
# Creating initiator and deleter

# Deleter

from stonesoup.deleter.error import CovarianceBasedDeleter
deleter = CovarianceBasedDeleter(covar_trace_thresh=4)

# Initiator

from stonesoup.initiator.simple import MultiMeasurementInitiator
initiator = MultiMeasurementInitiator(
    prior_state=prior1,
    measurement_model=None,
    deleter=deleter,
    data_associator=data_associator,
    updater=updater,
    min_points=2,
    )
```

The predicting and updating steps

We loop through each of our measurements and use the data associator to 'associate' the correct measurement to each track. It uses the 'generate_hypotheses()' method to return a list / MultipleHypothesis of SingleDistanceHypothesis data types for each measurement. It containins the prediction, detection and predicted measurement, as well as the distance (using Mahalanobis in this case). The GlobalNearestNeigbour takes the max of these from the valid Joint Hypotheses.

From this list, we then take the hypothesis for each track in the tracks set (any active tracks being updated). We then update the hypothesis using the ExtendedKalmanUpdater, and if there's no measurement, we keep the prediction.

We then run the deleter to remove any tracks which haven't been updated (where the state covariance exceeds the covariance trace threshold of 4), and we initiate any new tracks from unassociated measurements. These are firstly recorded as 'holding tracks', but if the number of states in the 'holding track' exceeds the min_points (specified as 2 above), then we add the track to our 'tracks' set.

We keep a record of all tracks which have been created / deleted.

In [17]:
```
from stonesoup.types.track import Track
# tracks = {Track([prior1]),Track([prior2])}

tracks, all_tracks = set(), set()
# test_set = set()
# list_prob_hiding = []
for n, measurements in enumerate(measurements_all):
    # Need single association
    hypotheses = data_associator.associate(tracks, measurements,
                                           start + timedelta(seconds=n))

    associated_measurements = set()
    for track in tracks:
```

```
        hypothesis = hypotheses[track]
#         test_set.add(track)

        if hypothesis.measurement:
            post = updater.update(hypothesis)
            track.append(post)
            associated_measurements.add(hypothesis.measurement)
        else:
            track.append(hypothesis.prediction)

        # pr(hiding) = f(measurement, location / co-ordinates) - need a metric which indicates
        # line path or into area of low visibility (known). Or fewer measurements than expected

    # Initiator and deleter

    tracks -= deleter.delete_tracks(tracks)
    tracks |= initiator.initiate(measurements - associated_measurements,
                                 start + timedelta(seconds=n))
    all_tracks |= tracks
```

Plotting the Output

Plotting 'all_tracks' using the Stone Soup Plotter, we have the below.

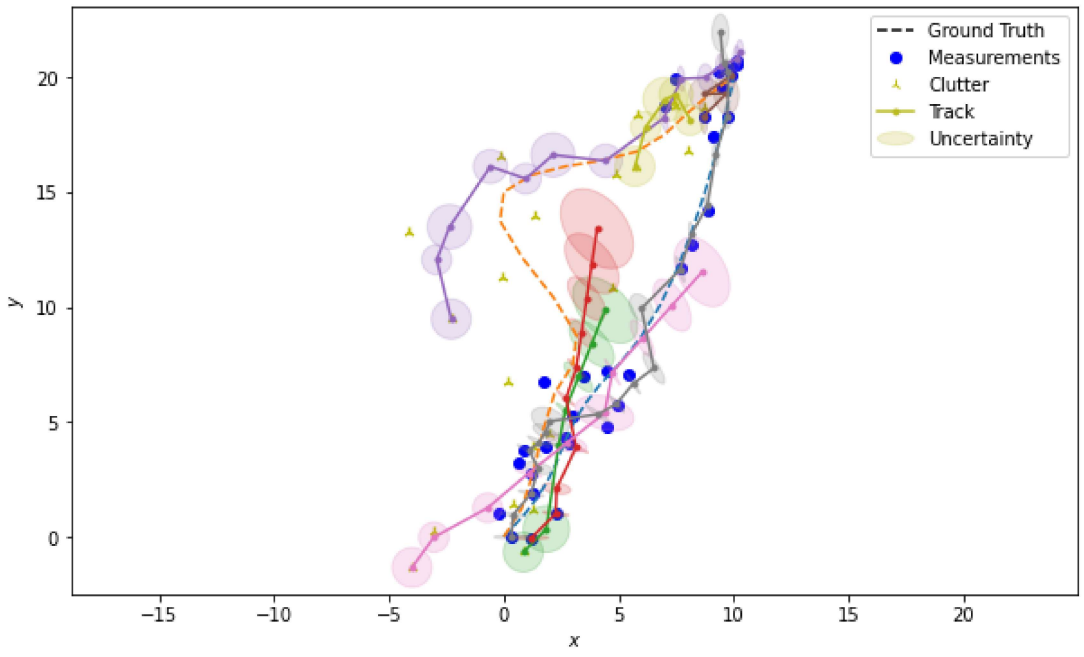
We begin with a track for both objects. However, the first object then deviates from its original path and becomes undetectable (assumed). The track is then updated until its uncertainty grows and is not associated to a new detection. Once the object comes back into view, a new track is initiated, and paired with the incoming detections. We see a second track picking this up at the end of the simulation. In fact, it appears that the track is actually paired to a clutter point before it is then joined to the new True Detection of the object coming back into the field of view.

The track undergoing no deceptive behaviour is mapped from the beginning to end waypoints.

In [18]:
```
plotter.plot_tracks(all_tracks,[0,2],uncertainty=True)
```

In [19]:
```
plotter.fig
```

Out[19]:



Task List:

- Initiator and deleter for deceptive track. DONE.
- Include clutter - data association problem. Done. Next fix the data association problem. DONE.
- Add in quadrant classification for deception e.g. when track is in certain co-ordinates. DONE.

- Next steps:

- Write up tutorial for the above for next thurs. DONE

- Look into metrics to infer prob(deception). How close inferred track is to ground truth. - check the Liverpool paper.
- Add in clutter even when there's no detections. - DONE
- 'Interacting multiple models': Check in with James : Take the inferences and assess against a given model - Probability that the track follows a given model (e.g. Constant Velocity etc.)
  - Check the simulator example in stone soup - (combined transition models)

In [ ]:

In [ ]:

In [ ]:

In [ ]: