Preview | Code | Blame | 777 lines | 29.7 KiB | Raw | Edit | History

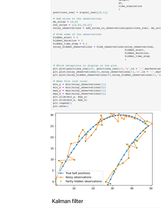
Implementing the trajectory of the ball throw with noisy and hidden observations

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

def generate_ball_in_air(num_steps,
                         noise_stddev=None):
    """
    Generate ground truth and optionally noisy ball throw data.
    :param num_steps: Number of time steps.
    :param noise_stddev: Standard deviation of Gaussian noise to add to positions.
    :return: (true_states, noisy_states) if noise_stddev is not None,
             (true_states, None) otherwise.
    """
    # Initial conditions
    x = 0.0
    y = 0.0
    vx = 10.0
    vy = 20.0
    g = 9.81
    dt = 0.1  # time step
    true_states = []
    for i in range(num_steps):
        true_states.append((x, y, vx, vy))
        x = x + vx * dt
        y = y + vy * dt
        vy = vy - g * dt
    true_states = np.array(true_states)

    if noise_stddev is None:
        return true_states, None

    noise = np.random.normal(0, noise_stddev, true_states.shape[:2])
    noisy_states = true_states + noise
    return true_states, noisy_states

def plot_trajectory(true_states, noisy_states=None):
    """
    Plot the ground truth and optionally noisy ball throw data.
    :param true_states: Array of shape (num_steps, 4) with [x, y, vx, vy].
    :param noisy_states: Array of shape (num_steps, 2) with [x, y] or None.
    :return: None.
    """
    plt.figure(figure_size=(8, 6))
    plt.plot(true_states[:, 0], true_states[:, 1], 'b-', label='True Ball Trajectory')
    if noisy_states is not None:
        plt.scatter(noisy_states[:, 0], noisy_states[:, 1], c='r', s=20, alpha=0.6, label='Noisy Observations')
    plt.xlabel('Position x')
    plt.ylabel('Position y')
    plt.title('Ball Trajectory')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Plotting the trajectory of the ball and the observations

In [2]:
```
# Implementation of the trajectory and observations
num_steps = 50
true_states, noisy_states = ball_throw_trajectories(num_steps=num_steps,
                                                    noise_stddev=0.5)
print(true_states.shape)
print(noisy_states.shape)
# Plot the trajectory and observations
plot_trajectory(true_states, noisy_states)
```

```
(50, 4)
(50, 2)
```



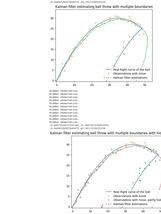
Kalman filter

In [3]:
```
class KalmanFilterTrajectory:
    def __init__(self, dt=0.1, initial_state=None, process_noise=1.0, measurement_noise=1.0):
        """
        Initialize the Kalman filter.
        :param dt: Time step.
        :param initial_state: Initial state [x, y, vx, vy].
        :param process_noise: Process noise.
        :param measurement_noise: Measurement noise.
        """
        self.dt = dt
        self.n = 4  # state dimension
        self.m = 2  # measurement dimension

        # Process noise covariance
        self.Q = np.eye(self.n) * process_noise

        # Measurement noise covariance
        self.R = np.eye(self.m) * measurement_noise

        # Initial state
        if initial_state is None:
            self.x = np.zeros(self.n)
            self.x = initial_state

        # Initial covariance
        self.P = np.eye(self.n) * 1.0

        # Measurement matrix (we only measure position)
        self.H = np.array([[1, 0, 0, 0],
                           [0, 1, 0, 0]])

        # Control input matrix (gravity)
        self.B = np.array([[0, 0, 0, 0],
                           [0, 0, 0, 0],
                           [0, 0, 1, 0],
                           [0, 0, 0, 1]])

        # Measurement noise covariance
        self.R = np.eye(self.m) * measurement_noise

        # Gravity vector
        self.g = np.array([0, 0, 0, -9.81])

    def predict(self, u=None):
        """
        Prediction step.
        :param u: Control input (if any).
        :return: Predicted state.
        """
        # Prediction
        if u is None:
            u = np.zeros(self.n)
        self.x = self.A @ self.x + self.B @ u + self.g
        self.P = self.A @ self.P @ self.A.T + self.Q

        return self.x

    def update(self, z):
        """
        Update step.
        :param z: Measurement.
        :return: Updated state.
        """
        # Kalman gain
        K = self.P @ self.H.T @ np.linalg.inv(self.H @ self.P @ self.H.T + self.R)

        # Update
        self.x = self.x + K @ (z - self.H @ self.x)
        self.P = (np.eye(self.n) - K @ self.H) @ self.P

        return self.x
```

Applying the Kalman filter to ball three with walls

In [4]:
```
noisy_value = 1000000
dt = 0.1
dt = 0.1
initial_state = [
initial_state = [0, 0, 10, 20]
process_noise = 0.1
measurement_noise = 0.5
num_steps = 50
true_states, noisy_states = ball_throw_trajectories(num_steps=num_steps,
                                                    noise_stddev=noisy_value)

hidden_states = ball_throw_trajectories(num_steps=num_steps,
                                        noise_stddev=noisy_value)
hidden_states = np.array(hidden_states)
hidden_states[:, 0] = np.nan
hidden_states[:, 1] = np.nan

kf = KalmanFilterTrajectory(dt=dt, initial_state=initial_state, process_noise=process_noise, measurement_noise=measurement_noise)

estimated_states = []
for i in range(num_steps):
    if i % 10 == 0:
        print(f"Step {i}, Using observation")
        kf.update(noisy_states[i])
    else:
        print(f"Step {i}, Skipping observation")
        kf.predict()
    estimated_states.append(kf.x.copy())
estimated_states = np.array(estimated_states)

# Plot results
plt.figure(figure_size=(8, 6))
plt.plot(true_states[:, 0], true_states[:, 1], 'b-', label='True Ball Trajectory')
plt.scatter(noisy_states[:, 0], noisy_states[:, 1], c='r', s=20, alpha=0.6, label='Noisy Observations')
plt.scatter(hidden_states[:, 0], hidden_states[:, 1], c='g', s=20, alpha=0.6, label='Hidden Observations')
plt.plot(estimated_states[:, 0], estimated_states[:, 1], 'k--', label='Kalman Filter Estimate')
plt.xlabel('Position x')
plt.ylabel('Position y')
plt.title('Kalman Filter Estimating Ball Throw with Multiple Boundaries')
plt.legend()
plt.grid(True)
plt.show()
```

```
Step 0, Using observation
Step 1, Skipping observation
Step 2, Skipping observation
Step 3, Skipping observation
Step 4, Skipping observation
Step 5, Skipping observation
Step 6, Skipping observation
Step 7, Skipping observation
Step 8, Skipping observation
Step 9, Skipping observation
Step 10, Using observation
Step 11, Skipping observation
Step 12, Skipping observation
Step 13, Skipping observation
Step 14, Skipping observation
Step 15, Skipping observation
Step 16, Skipping observation
Step 17, Skipping observation
Step 18, Skipping observation
Step 19, Skipping observation
Step 20, Using observation
Step 21, Skipping observation
Step 22, Skipping observation
Step 23, Skipping observation
Step 24, Skipping observation
Step 25, Skipping observation
Step 26, Skipping observation
Step 27, Skipping observation
Step 28, Skipping observation
Step 29, Skipping observation
Step 30, Using observation
Step 31, Skipping observation
Step 32, Skipping observation
Step 33, Skipping observation
Step 34, Skipping observation
Step 35, Skipping observation
Step 36, Skipping observation
Step 37, Skipping observation
Step 38, Skipping observation
Step 39, Skipping observation
Step 40, Using observation
Step 41, Skipping observation
Step 42, Skipping observation
Step 43, Skipping observation
Step 44, Skipping observation
Step 45, Skipping observation
Step 46, Skipping observation
Step 47, Skipping observation
Step 48, Skipping observation
Step 49, Skipping observation
```



Kalman filter estimating ball throw with multiple boundaries with hidden observations

Calculating the error of estimation for specific parts of the throw (for the paper)

In [5]:
```
# Calculating estimation error
distances = []
for i in range(num_steps):
    if i % 10 == 0:
        pass
    else:
        x_estimate = estimated_states[i, 0]
        y_estimate = estimated_states[i, 1]
        x_true = true_states[i, 0]
        y_true = true_states[i, 1]
        distance = np.sqrt((x_estimate - x_true)**2 + (y_estimate - y_true)**2)
        distances.append(distance)
print("Average error:", distances)
```

```
Average error: [nan, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```