Exercise_round_8

March 13, 2025

0.1 Exercise 1. (Multi-modal posterior)

Consider the following partially observed random-walk model:

$$p(x_k|x_{k-1}) = N(x_k|x_{k-1}, 1)p(y_k|x_k) = \text{LogNormal}(y_k|\log|x| - \frac{1}{2}, 1)$$

where $\mathcal{N}(x|\mu, \sigma^2)$ denotes the normal distribution with mean μ and variance σ^2.

where the log-normal distribution density function is:

$$\text{LogNormal}(x|a, b) \propto \exp\left(-\frac{(\log(x) - a)^2}{2b^2}\right)$$

In this case, we have $a = \frac{1}{2}$ and $b = 1$.

Hint: The log-normal distribution is defined such that $\mathbb{E}[y_k|x_k] = |x_k|$.

Take x_0 to have the standard normal distribution. Generate synthetic data from this model and apply

(a) one of the Gaussian approximation-based filters,

(b) a bootstrap particle filter using 1,000 particles.

Compare the posterior distributions given by the two methods above: for the Gaussian filter, plot the mean and uncertainty (0.95 standard deviation), and for the bootstrap filter, show a scatter plot of the filtering distribution at each time step.

[76]:
```
# Necessary imports
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, lognorm
```

Let us use Kalman Filter as an example of a Gaussian-based filter. And compare it to a Bootstrap Particle Filter with 1000 particles.

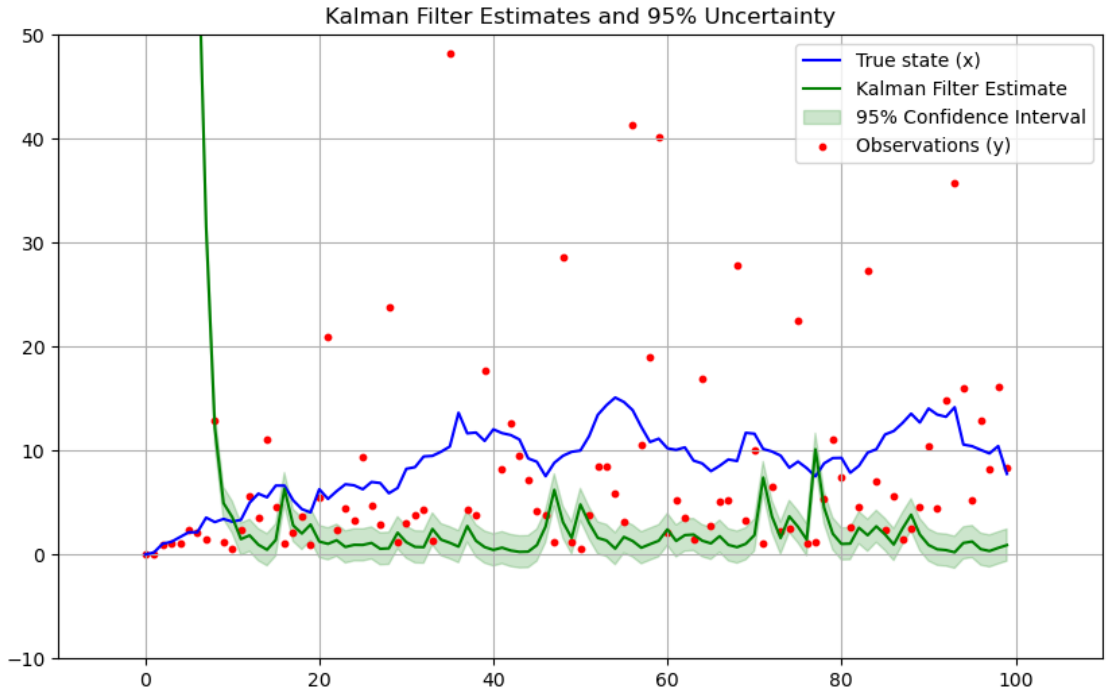
[77]:
```
# Extendede Kalman Filter
def EKF_1(observations, m_0, P_0, R, Q):
    # Initialize state and covariance
    m = m_0
    P = P_0
```

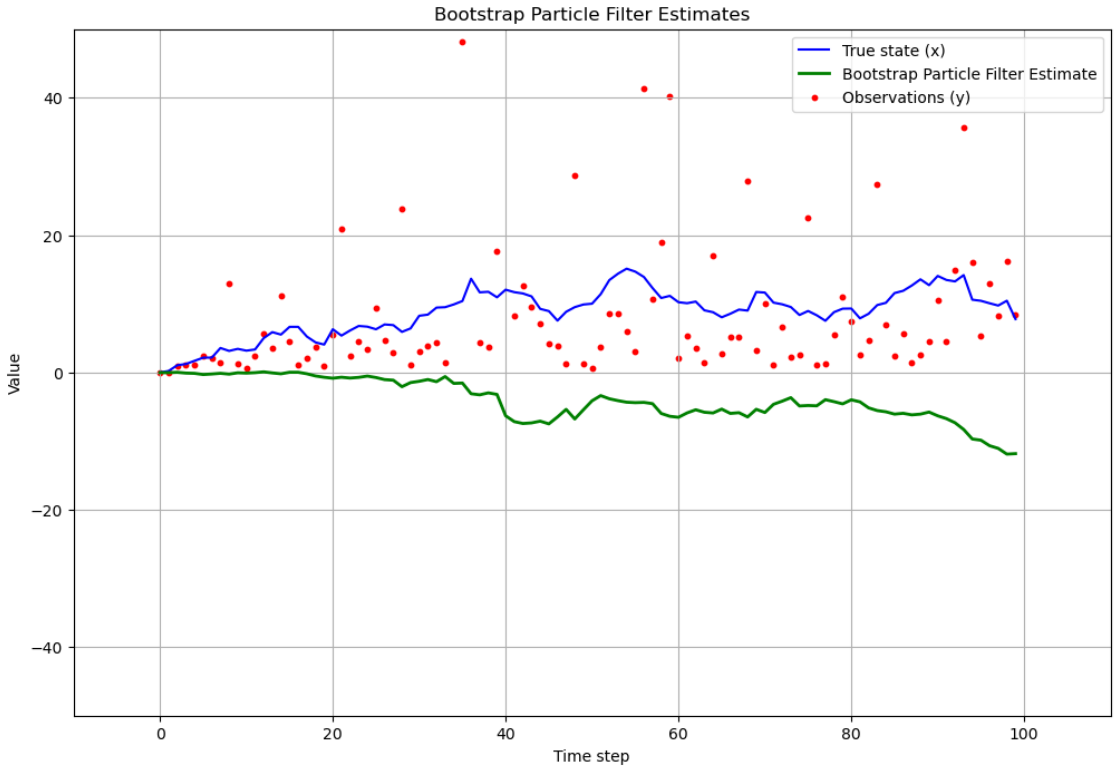
```
    # Kalman filter loop
    m_estimates = []
    P_estimates = []

    for obs in observations:
        # Prediction Step
        m_pred = m  # Predicted state estimate
        P_pred = P + Q  # Predicted covariance

        # Update Step
        obs = h_jacobian(obs) # Linearize the measurement model
        y = obs - m_pred  # Innovation
        S = P_pred + R  # Innovation covariance
        K = P_pred * 1/S  # Kalman gain

        m = m_pred + K * y  # Updated state estimate
        P = P_pred - K ** 2 * S  # Updated covariance

        # Save estimate
        m_estimates.append(m)
        P_estimates.append(P)

    return np.array(m_estimates), np.array(P_estimates)

# Particle Filter (Bootstrap)
def particle_filter_1(num_particles, T, x_true, y_obs):
    # Initialize particles and weights
    particles = np.random.normal(0, 1, num_particles)  # Initial particles␣
 ↪(Gaussian)
    weights = np.ones(num_particles) / num_particles  # Uniform initial weights

    # Arrays to store estimates
    estimates = np.zeros(T)

    for t in range(T):
        # Prediction step: Sample from transition model for each particle
        particles = np.array([transition(p) for p in particles])

        # Weighting step: Compute weights based on observation likelihood
        predicted_observations = np.array([observation(p) for p in particles])

        # Avoid log(0) and handle invalid values in likelihood
        predicted_observations = np.maximum(predicted_observations, 1e-10)  #␣
 ↪Prevent zero or negative values in log

        # Likelihood of each observation given the predicted state
```

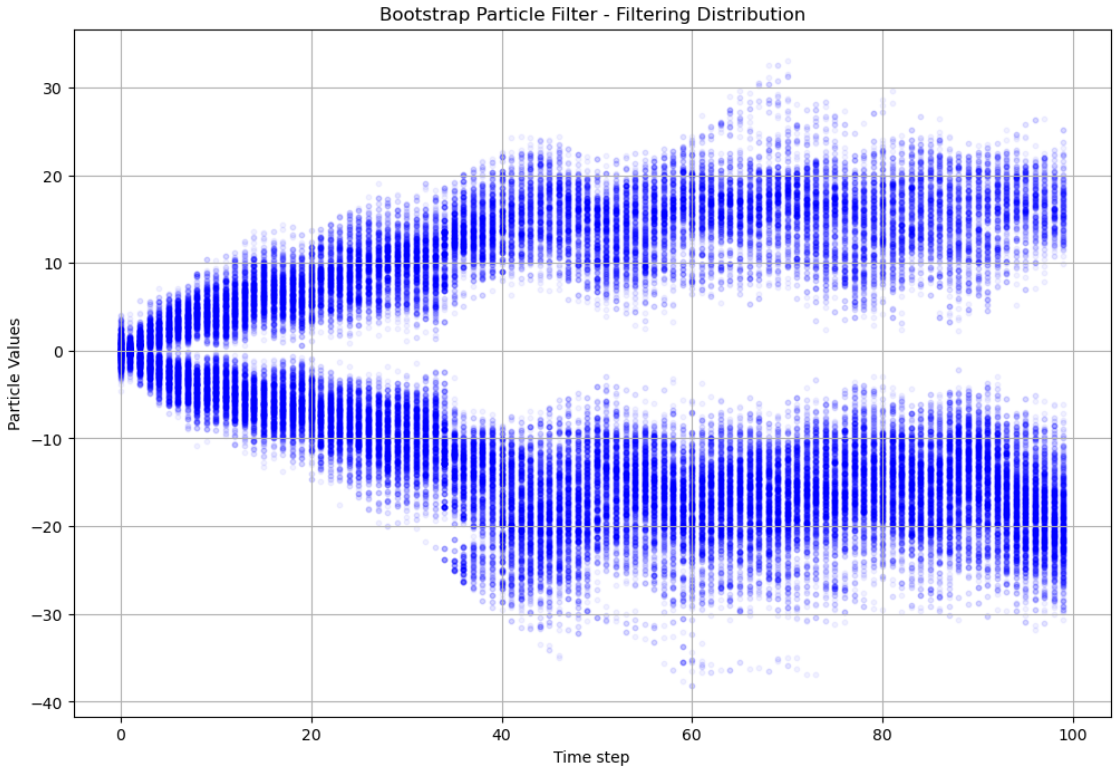
```
        likelihoods = np.exp(-0.5 * (np.log(np.abs(predicted_observations)) -␣
↪np.log(np.abs(y_obs[t]+1e-10)) - 0.5)**2)

        # Prevent weights from becoming zero or NaN
        weights = weights * likelihoods
        weights += 1e-10  # Add a small value to avoid zero weights
        weights /= np.sum(weights)  # Normalize the weights


        # Resampling step: Resample particles according to the updated weights
        indices = np.random.choice(range(num_particles), size=num_particles,␣
↪p=weights)
        particles = particles[indices]
        weights = np.ones(num_particles) / num_particles  # Reset weights after␣
↪resampling

        # Estimate the state: Take the weighted mean of the particles
        estimates[t] = np.mean(particles)

        particles_bpf[t, :] = particles

    return estimates

np.random.seed(68)

# Model Parameters
T = 100  # Time steps
x0 = np.random.normal(0, 1)  # Initial state x0 ~ N(0, 1)

Q = 1.0 # process noise
R = 1.0 # measurement noise

num_particles = 1000 # Particles

# Generate the true states (x_k) based on the random walk model
x_true = np.zeros(T)
observations = np.zeros(T)

# Define the system model (Random walk)
def transition(x):
    return x + np.random.normal(0, Q)

# Define the observation model (Log-Normal observation)
def observation(x):
    return np.exp(np.log(np.abs(x) + 1e-10) - 0.5 + np.random.normal(0, R))

def h_jacobian(x):
```

```
    return np.exp(1/(x+0.1) - 0.5 + np.random.normal(0, R))  # Jacobian of h(x)


# Simulate the process over time
for t in range(1, T):
    x_true[t] = transition(x_true[t - 1])  # Transition step: random walk
    observations[t] = observation(x_true[t])  # Observation step: Log-Normal


# Apply Extended Kalman Filter
m0 = x0
P0 = 0.1
m_ekf, P_ekf = EKF_1(observations, m0, P0, R, Q)

m_ekf = m_ekf.flatten()
P_ekf = P_ekf.flatten()

# Apply Boostrap Particle Filter
particles_bpf = np.zeros((T, num_particles))
x_bpf = particle_filter_1(num_particles, T, x_true, observations)

# Plot results for KF
plt.figure(figsize=(10, 6))
plt.grid(True)
plt.plot(x_true, label="True state (x)", color='blue')
plt.plot(m_ekf, label="Kalman Filter Estimate", color='green')
plt.fill_between(range(T), m_ekf - 1.96 * np.sqrt(P_ekf),
                 m_ekf + 1.96 * np.sqrt(P_ekf),
                 color='green', alpha=0.2, label="95% Confidence Interval")
plt.scatter(range(T), observations, label="Observations (y)", color='red', s=10)
plt.title("Kalman Filter Estimates and 95% Uncertainty")
plt.xlim(T-(T+10), T+10)
plt.ylim(-10,50)
plt.legend()
plt.show()


# Plot results for BPF
plt.figure(figsize=(12, 8))
plt.grid(True)
plt.plot(x_true, label="True state (x)", color='blue')
plt.plot(x_bpf, label="Bootstrap Particle Filter Estimate", color='green',␣
 ↪linewidth=2)
plt.title("Bootstrap Particle Filter Estimates")
plt.scatter(range(T), observations, label="Observations (y)", color='red', s=10)
plt.xlabel("Time step")
plt.ylabel("Value")
```

```
plt.xlim(T-(T+10), T+10)
plt.ylim(-50, 50)
plt.legend()
plt.show()

plt.figure(figsize=(12, 8))
plt.grid(True)

for t in range(T):
    # Scatter plot for the particles at time step t
    plt.scatter(np.full(num_particles, t), particles_bpf[t, :], label=f'Time␣
 ↪step {t}', alpha=0.05, color='blue', s=10)

plt.xlabel("Time step")
plt.ylabel("Particle Values")
plt.title("Bootstrap Particle Filter - Filtering Distribution")
plt.show()
```




Bootstrap Particle Filter Estimates
True state (x)
Bootstrap Particle Filter Estimate
Observations (y)
Value
Time step
40
20
0
−20
−40
0
20
40
60
80
100


Bootstrap Particle Filter - Filtering Distribution
Particle Values
Time step
30
20
10
0
−10
−20
−30
−40
0
20
40
60
80
100

0.2 Exercise 2. (Locally Optimal Proposal and Run Time)

Consider the following linear Gaussian state space model:

$$\mathrm{x}_k = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \mathrm{x}_{k-1} + \mathrm{q}_{k-1}$$

$$y_k = \begin{pmatrix} 1 & 0 \end{pmatrix} \mathrm{x}_k + r_k$$

where $\mathrm{x}_k = (x_k, \dot{x}_k)$ is the state, y_k is the measurement, and $\mathrm{q}_k \sim N(0, \mathrm{diag}(1/10^2, 1))$ and $r_k \sim N(0, 10^2)$ are white Gaussian noise processes. x_0 has a standard normal distribution.

(a) Derive an expression for the optimal importance distribution for the model:

$$\pi(\mathrm{x}_k) = p(\mathrm{x}_k | \mathrm{x}_{k-1}, \mathrm{y}_{1:k})$$

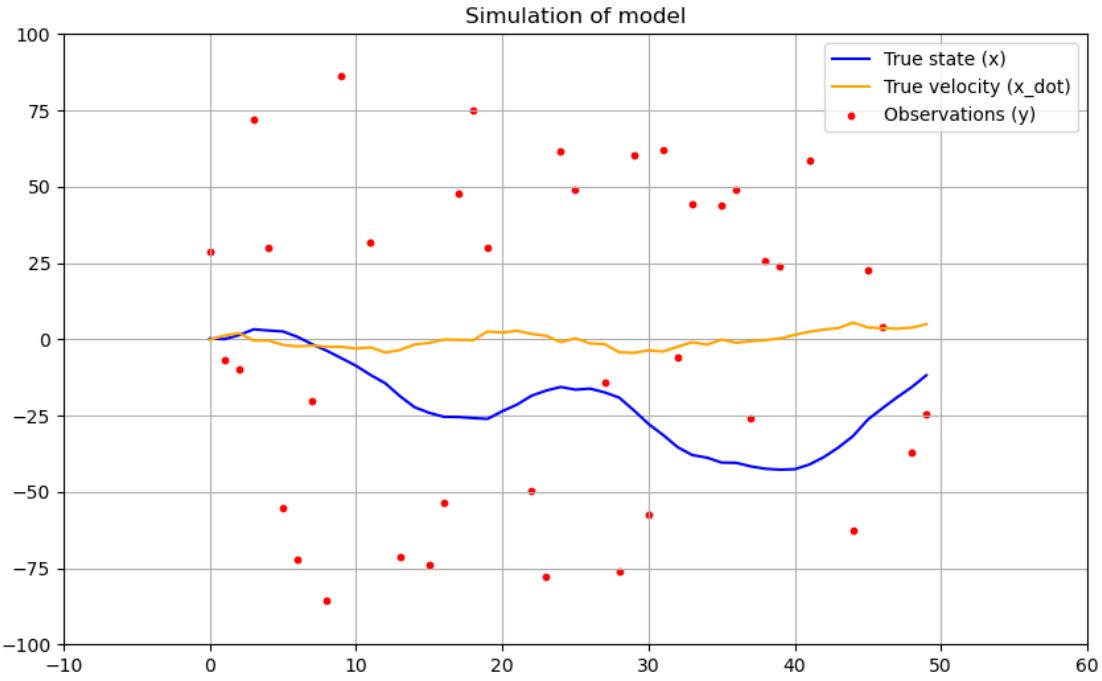
(b) Simulate some data from the model for $T = 50$ time steps and implement a Kalman filter for it.

(c) Implement a bootstrap particle filter for the model. Implement it using vectorized code such that it is as fast as possible.

(d) Measure the run time taken by each algorithm. Can you get the particle filter to be as fast as the Kalman filter for any number of particles?

[78]:
```
# initialization

np.random.seed(99)

x0 = np.array([[0], [0]])

Q = np.diag([1/10**2, 1**2])
R = 10**2

F = np.array([[1,1],[0,1]])
H = np.array([[1,0]])

# generate simulation

steps = 50

x_true = np.zeros((steps, 2))
observations = np.zeros(steps)

for k in range(steps):
    x_true[k] = F @ x_true[k-1] + np.random.multivariate_normal([0, 0],Q)
    observations[k] = H @ x_true[k] + np.random.normal(0,R)

# Plot simulation
```

```
plt.figure(figsize=(10, 6))
plt.grid(True)

plt.plot(x_true[:,0], label="True state (x)", color='blue')
plt.plot(x_true[:,1], label="True velocity (x_dot)", color='orange')
plt.scatter(range(steps), observations, label="Observations (y)", color='red',␣
↪s=10)

plt.title("Simulation of model")
plt.xlim(steps-(steps+10), steps+10)
plt.ylim(-100, 100)
plt.legend(loc='upper right')
plt.show()
```

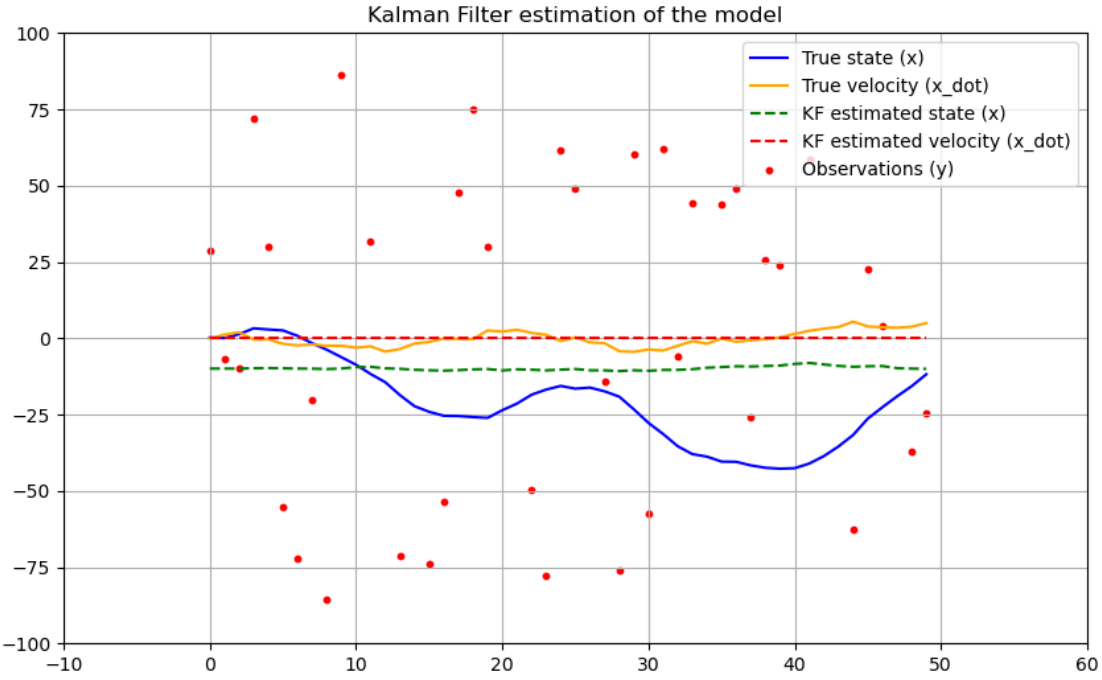


[79]:
```
# Kalman Filter
def Kalman_Filter_2(observations, m_0, P_0, F, H, Q, R):
    # Initialize state and covariance
    m = m_0
    P = P_0

    # Kalman filter loops
    m_estimates = []
    P_estimates = []

    for obs in observations:
```

```
        # Prediction Step
        m_pred = F @ m  # Predicted state estimate
        P_pred = F @ P @ np.linalg.inv(F) + Q  # Predicted covariance

        # Update Step
        y = obs - H @ m_pred  # Innovation
        S = H @ P_pred @ H.T + R  # Innovation covariance
        K = P_pred @ H.T @ np.linalg.inv(S)  # Kalman gain

        m = m_pred + K @ y  # Updated state estimate
        P = P_pred - K @ S @ K.T  # Updated covariance

        # Save estimate
        m_estimates.append(m)
        P_estimates.append(P)

    m_estimates = np.array(m_estimates)
    P_estimates = np.array(P_estimates)

    return m_estimates, P_estimates
```

[80]:
```
import time

# run the Kalman filter on the model (with timing)
m0 = np.array([[-10], [0]])
P0 = np.diag([0.1, 0.1])

start_kf = time.time()
x_kf, P_kf = Kalman_Filter_2(observations, m0, P0, F, H, Q, R)
end_kf = time.time()
time_kf = end_kf - start_kf

# Plot KF results
plt.figure(figsize=(10, 6))
plt.grid(True)

plt.plot(x_true[:,0], label="True state (x)", color='blue')
plt.plot(x_true[:,1], label="True velocity (x_dot)", color='orange')

# Plot Kalman filter estimates
plt.plot(x_kf[:, 0], label="KF estimated state (x)", linestyle='--',␣
 ↪color='green')  # Estimated position
plt.plot(x_kf[:, 1], label="KF estimated velocity (x_dot)", linestyle='--',␣
 ↪color='red')  # Estimated velocity

plt.scatter(range(steps), observations, label="Observations (y)", color='red',␣
 ↪s=10)
```

```
plt.title("Kalman Filter estimation of the model")
plt.xlim(steps-(steps+10), steps+10)
plt.ylim(-100, 100)
plt.legend(loc='upper right')
plt.show()
```

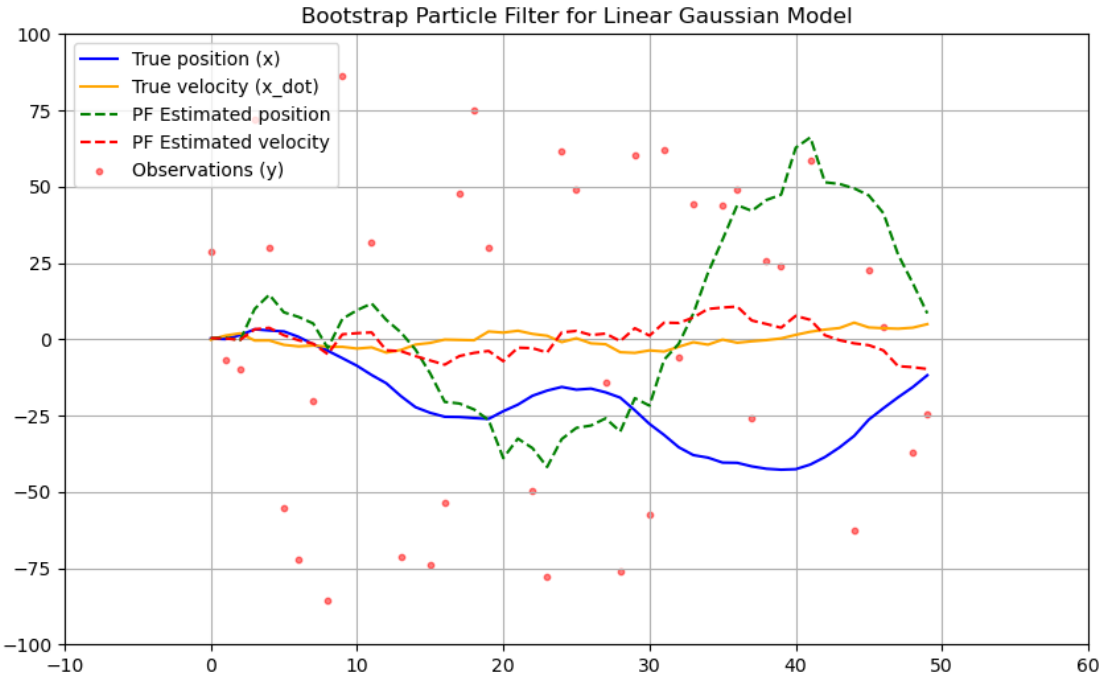


[81]:
```
#pip install filterpy
```

[82]:
```
#from filterpy.monte_carlo import systematic_resample

# Bootstrap Particle Filter
def Particle_Filter_2(observations, num_particles, F, H, Q, R):

    steps = len(observations)

    # Initialize particles and weights
    particles = np.random.multivariate_normal([0, 0], Q, size=num_particles)  #␣
 ↪Prior particles
    weights = np.ones(num_particles) / num_particles  # Equal initial weights

    # Storage for estimates
    x_estimates = np.zeros((steps, 2))

    for k in range(steps):
        # Prediction step: Propagate particles
```

```
        process_noise = np.random.multivariate_normal([0, 0], Q, num_particles)
        particles = (F @ particles.T).T + process_noise  # Vectorized transition

        # Compute weights using observation likelihood
        y_pred = (H @ particles.T).T  # Predicted measurements
        weights = np.exp(-0.5 * ((observations[k] - y_pred) ** 2) / R)  #␣
↪Gaussian likelihood
        weights /= np.sum(weights)  # Normalize

        # Estimate state as weighted mean of particles
        x_estimates[k, :] = np.sum(particles * weights[:], axis=0)  # Weighted␣
↪mean

        # Resampling
        #indices = systematic_resample(weights) # faster resampling
        indices = np.random.choice(np.arange(num_particles),␣
↪size=num_particles, p=weights.flatten())
        particles = particles[indices]  # Resample particles
        weights.fill(1.0 / num_particles)  # Reset weights

    return x_estimates, particles
```

[83]:
```
# run the BPF on the model (with timing)
num_particles = 1000

start_bpf = time.time()
x_bpf, particle_bpf = Particle_Filter_2(observations, num_particles, F, H, Q, R)
end_bpf = time.time()
time_bpf = end_bpf - start_bpf

# Plot BPF estimates
plt.figure(figsize=(10, 6))
plt.plot(x_true[:, 0], label="True position (x)", color='blue')
plt.plot(x_true[:, 1], label="True velocity (x_dot)", color='orange')
plt.plot(x_bpf[:, 0], label="PF Estimated position", linestyle="--",␣
↪color='green')
plt.plot(x_bpf[:, 1], label="PF Estimated velocity", linestyle="--",␣
↪color='red')
plt.scatter(range(steps), observations, label="Observations (y)", color='red',␣
↪s=10, alpha=0.5)
plt.xlim(steps-(steps+10), steps+10)
plt.ylim(-100,100)
plt.legend()
plt.title("Bootstrap Particle Filter for Linear Gaussian Model")
plt.grid(True)
plt.show()
```



[84]:
```
print("Timings:\n KF:", time_kf*1000,"ms \n BPF:", time_bpf*1000, "ms")
```

```
Timings:
 KF: 5.545377731323242 ms
 BPF: 34.41882133483887 ms
```

No other method for BPF that is much faster after vectorizing, than just reducing the number of particles.

0.3 Exercise 3

For the same state-space model in equation (3), compute the empirical mean and variance of the weighted particles returned by the particle filter, and compare them against the Kalman filter means and covariances in the following situations (you can look at just the final time step):

1. For number of particles $N \in 10^2, 10^3, 10^4, 10^5$.
2. (a) No resampling (b) Resample at every time step (c) Resample only when the effective sample size is below 75% (choose $N = 10^3$ for this part).

[85]:
```
# 1.
x_bpf_102, particles_bpf_102 = Particle_Filter_2(observations, 10**2, F, H, Q,␣
 ↪R)
x_bpf_103, particles_bpf_103 = Particle_Filter_2(observations, 10**3, F, H, Q,␣
 ↪R)
x_bpf_104, particles_bpf_104 = Particle_Filter_2(observations, 10**4, F, H, Q,␣
 ↪R)
```

```
x_bpf_105, particles_bpf_105 = Particle_Filter_2(observations, 10**5, F, H, Q,␣
 ↪R)

# Compute mean and variance of weighted particles
p_102_avg = np.sum(particles_bpf_102, axis=0) / 100
p_102_var = np.var(particles_bpf_102, axis=0)

p_103_avg = np.sum(particles_bpf_103, axis=0) / 1000
p_103_var = np.var(particles_bpf_103, axis=0)

p_104_avg = np.sum(particles_bpf_104, axis=0) / 10000
p_104_var = np.var(particles_bpf_104, axis=0)

p_105_avg = np.sum(particles_bpf_105, axis=0) / 100000
p_105_var = np.var(particles_bpf_105, axis=0)
```

[86]:
```
# Compare the BPF averages and variances with the KF estimates
print("KF final states and covariance:")
print("states:\n", x_kf[-1], "\n")
print("covariances:\n", P_kf[-1], "\n")
print("particle averages and variances:")
print("N = 10^2:", p_102_avg, p_102_var)
print("N = 10^3:", p_103_avg, p_103_var)
print("N = 10^4:", p_104_avg, p_104_var)
print("N = 10^5:", p_105_avg, p_105_var)
```

```
KF final states and covariance:
states:
 [[-10.04151957]
 [  0.        ]]

covariances:
 [[5.35877114e-01 1.21371843e+03]
 [0.00000000e+00 5.01000000e+01]]

particle averages and variances:
N = 10^2: [61.36048171 -1.43592516] [0.94265147 1.41477678]
N = 10^3: [64.67157507 -3.47377122] [1.48658773 2.46137775]
N = 10^4: [13.8219172  -8.84444025] [4.05048145 2.70077631]
N = 10^5: [-44.4457105  -13.73252564] [16.36120495  3.46789939]
```