**Real-Time Radar Target Tracking and Classification System**

**Abstract** — This report provides a detailed analysis of a Python-based system designed for real-time 2D/3D radar target tracking and classification. The system integrates classical state estimation techniques, namely the **Kalman Filter** for linear motion and the **Particle Filter** for non-linear dynamics, with modern machine learning, specifically **Support Vector Machines (SVM)** and **Random Forest Classifiers**, for motion-based target classification. The report covers the simulation of ground truth and noisy radar measurements, the implementation of both filters, a comparative analysis of their tracking performance using RMSE, and the methodology for feature extraction and classification using synthetic motion data. Visualizations including 3D tracking plots, per-step error curves, and confusion matrices are presented to illustrate system performance. The project serves as a robust educational and practical tool for understanding radar signal processing and its applications.

**I. INTRODUCTION**

Radar systems are indispensable in a myriad of modern applications, ranging from autonomous navigation and air traffic control to weather forecasting and defense. A core capability of these systems lies in their ability to accurately **track** moving objects and **classify** them based on their observed characteristics. This project addresses these fundamental aspects by developing a Python-based simulation that demonstrates real-time radar target tracking and classification.

The system's design emphasizes both accuracy in state estimation and intelligence in target identification. It leverages established filtering algorithms to handle noisy sensor data and employs supervised learning models to infer target types from their motion patterns. This report will systematically explore the various components of the system, their underlying principles, implementation details, and performance analysis based on simulated data.

**II. SYSTEM ARCHITECTURE AND COMPONENTS**

The project is structured into distinct phases, each managed by specific code segments within a unified Python script. This modularity facilitates clarity and allows for independent testing and refinement of each component.

**A. System Configuration**

The system begins with a set of configurable parameters that define the simulation environment and algorithm settings. These include T (time step), N (number of frames), true\_vel (ground truth constant velocity), meas\_noise\_std (radar measurement standard deviation), and parameters specific to the Kalman and Particle Filters, such as process\_var\_base, pf\_num\_particles, and pf\_vel\_noise.

Python

# ----------------------------------------------------------------

# CONFIG

# ----------------------------------------------------------------

np.random.seed(42) # for reproducibility

T = 1.0 # time step

N = 50 # number of frames

true\_vel = np.array([1.0, 0.5, 0.2]) # m/step constant velocity

process\_var\_base = 0.01 # KF base process noise

meas\_noise\_std = 2.0 # radar measurement std (meters)

pf\_num\_particles = 1000 # PF particle count

pf\_vel\_noise = 0.05 # PF velocity diffusion scale

use\_rf\_compare = True # also run RandomForest comparison

do\_grid\_svm = True # grid search SVM hyperparams

**B. Ground Truth and Measurement Simulation**

The initial phase involves simulating the **ground truth** trajectory of a target and generating **noisy radar measurements**. The target is assumed to move with a constant velocity in a 3D space. Gaussian noise, with a specified standard deviation (meas\_noise\_std), is added to the true positions to simulate realistic radar measurement inaccuracies.

Python

# ----------------------------------------------------------------

# PHASE 1: SIMULATE GROUND TRUTH & MEASUREMENTS

# ----------------------------------------------------------------

pos = np.zeros(3)

true\_pos = []

for \_ in range(N):

pos = pos + true\_vel

true\_pos.append(pos.copy())

true\_pos = np.array(true\_pos)

# Noisy radar measurements (Cartesian already)

measurements = true\_pos + np.random.normal(0, meas\_noise\_std, true\_pos.shape)

**III. RADAR TRACKING ALGORITHMS**

The core of the system's tracking capability relies on two distinct filtering techniques, chosen for their suitability in different motion scenarios.

**A. 3D Kalman Filter Tracking**

The **Kalman Filter (KF)** is implemented for tracking targets assumed to follow a **Constant Velocity (CV)** model in 3D space. The state vector for the KF is 6-dimensional, comprising position (x, y, z) and velocity (vx, vy, vz). The measurement model, H, maps the 6D state to 3D position measurements.

The KF operates in a predict-update cycle. In the **prediction step**, the filter projects the current state estimate and uncertainty covariance forward in time using the system's dynamic model (F) and process noise (Q). An adaptive component is introduced to Q to account for potential unmodeled maneuvers or increased uncertainty at each step. In the **update step**, the predicted state is corrected using the incoming radar measurement (z) and the measurement noise covariance (R), yielding a refined state estimate and reduced uncertainty.

Python

# ----------------------------------------------------------------

# PHASE 1 (cont.): 3D KALMAN FILTER TRACKING (6D CV MODEL)

# State: [x y z vx vy vz]^T, meas: [x y z]

# ----------------------------------------------------------------

F = np.block([

[np.eye(3), T\*np.eye(3)],

[np.zeros((3,3)), np.eye(3)]

])

H = np.block([

[np.eye(3), np.zeros((3,3))]

])

Q\_base = process\_var\_base \* np.eye(6)

R = (meas\_noise\_std\*\*2) \* np.eye(3)

x\_est = np.zeros((6,1)) # init state

P = np.eye(6) \* 100.0 # large initial uncertainty

kf\_estimates = []

kf\_vel\_history = [] # to extract motion features later

for z in measurements:

# Adaptive (small) jitter to Q each step to reflect maneuver uncertainty

adaptive\_Q = Q\_base + np.diag(np.abs(np.random.normal(0, process\_var\_base\*0.5, size=6)))

# Predict

x\_pred = F @ x\_est

P\_pred = F @ P @ F.T + adaptive\_Q

# Update

z = z.reshape(3,1)

y = z - H @ x\_pred

S = H @ P\_pred @ H.T + R

K = P\_pred @ H.T @ np.linalg.inv(S)

x\_est = x\_pred + K @ y

P = (np.eye(6) - K @ H) @ P\_pred

kf\_estimates.append(x\_est[:3].flatten())

kf\_vel\_history.append(x\_est[3:6].flatten())

kf\_estimates = np.array(kf\_estimates)

kf\_vel\_history = np.array(kf\_vel\_history)

# KF error metrics

kf\_errors = kf\_estimates - true\_pos

kf\_rmse = np.sqrt(np.mean(kf\_errors\*\*2, axis=1)) # per-step

kf\_rmse\_total = np.sqrt(np.mean(kf\_errors\*\*2)) # overall

**B. Particle Filter Tracking**

For scenarios involving potentially **non-linear motion**, a **Particle Filter (PF)** is implemented. Each particle in the PF represents a possible state (position and velocity) of the target. The PF operates through prediction, update, and resampling steps.

* **Prediction:** Particles are propagated forward in time based on a constant velocity model with added adaptive motion noise. This noise component adapts to the spread of particles, preventing degeneracy.
* **Update:** The likelihood of each particle's state is calculated based on its proximity to the current radar measurement. Weights are assigned to particles, with higher weights indicating better agreement with the measurement.
* **Resampling:** A **systematic resampling** method is used to eliminate particles with low weights and duplicate particles with high weights, ensuring that the particle set continues to represent the target's state effectively.

Python

# ----------------------------------------------------------------

# PHASE 2: PARTICLE FILTER (3D) WITH SYSTEMATIC RESAMPLING

# State per particle: [x y z vx vy vz]

# ----------------------------------------------------------------

def systematic\_resample(weights: np.ndarray) -> np.ndarray:

Np = len(weights)

positions = (np.arange(Np) + np.random.rand()) / Np

cumsum = np.cumsum(weights)

idx = np.zeros(Np, dtype=int)

i = j = 0

while i < Np:

if positions[i] < cumsum[j]:

idx[i] = j

i += 1

else:

j += 1

return idx

# init particles around first measurement

particles = np.zeros((pf\_num\_particles, 6))

particles[:, :3] = measurements[0] + np.random.normal(0, 5.0, (pf\_num\_particles, 3))

particles[:, 3:] = np.random.normal(0, 1.0, (pf\_num\_particles, 3))

weights = np.ones(pf\_num\_particles) / pf\_num\_particles

pf\_estimates = []

pf\_vel\_history = []

for t in range(N):

# Prediction: constant velocity + adaptive noise

# Increase motion\_noise when particle spread is small to avoid degeneracy

spread = np.mean(np.std(particles[:, :3], axis=0))

motion\_noise\_pos = np.clip(spread\*0.05, 0.01, 1.5)

particles[:, :3] += particles[:, 3:]\*T + np.random.normal(0, motion\_noise\_pos, (pf\_num\_particles, 3))

particles[:, 3:] += np.random.normal(0, pf\_vel\_noise, (pf\_num\_particles, 3))

# Update: measurement likelihood

z = measurements[t]

dists = np.linalg.norm(particles[:, :3] - z, axis=1)

weights = np.exp(-0.5 \* (dists / meas\_noise\_std)\*\*2)

weights += 1e-300

weights /= np.sum(weights)

# Resample

res\_idx = systematic\_resample(weights)

particles = particles[res\_idx]

weights = np.ones\_like(weights) / pf\_num\_particles

# Estimate (weighted mean -- uniform after resample so just mean)

est\_pos = np.mean(particles[:, :3], axis=0)

est\_vel = np.mean(particles[:, 3:], axis=0)

pf\_estimates.append(est\_pos)

pf\_vel\_history.append(est\_vel)

pf\_estimates = np.array(pf\_estimates)

pf\_vel\_history = np.array(pf\_vel\_history)

# PF error metrics

pf\_errors = pf\_estimates - true\_pos

pf\_rmse = np.sqrt(np.mean(pf\_errors\*\*2, axis=1))

pf\_rmse\_total = np.sqrt(np.mean(pf\_errors\*\*2))

**IV. PERFORMANCE VISUALIZATION**

Visualizations are critical for understanding the performance of the tracking algorithms. The project generates 3D plots of the trajectories and a 2D plot of the per-step tracking error.

**A. 3D Tracking Trajectories**

Figure 1 displays the 3D trajectories for both the Kalman Filter and Particle Filter. The **True Position** (blue line) represents the actual path of the simulated target. The **Radar Measurements** (red 'x' marks) show the noisy observations received by the radar system. The **Kalman Estimate** (green line) and **Particle Filter Estimate** (purple line) illustrate the smoothed and estimated paths computed by each filter, respectively. The RMSE (Root Mean Squared Error) values are provided in the titles for a quick assessment of overall tracking accuracy.

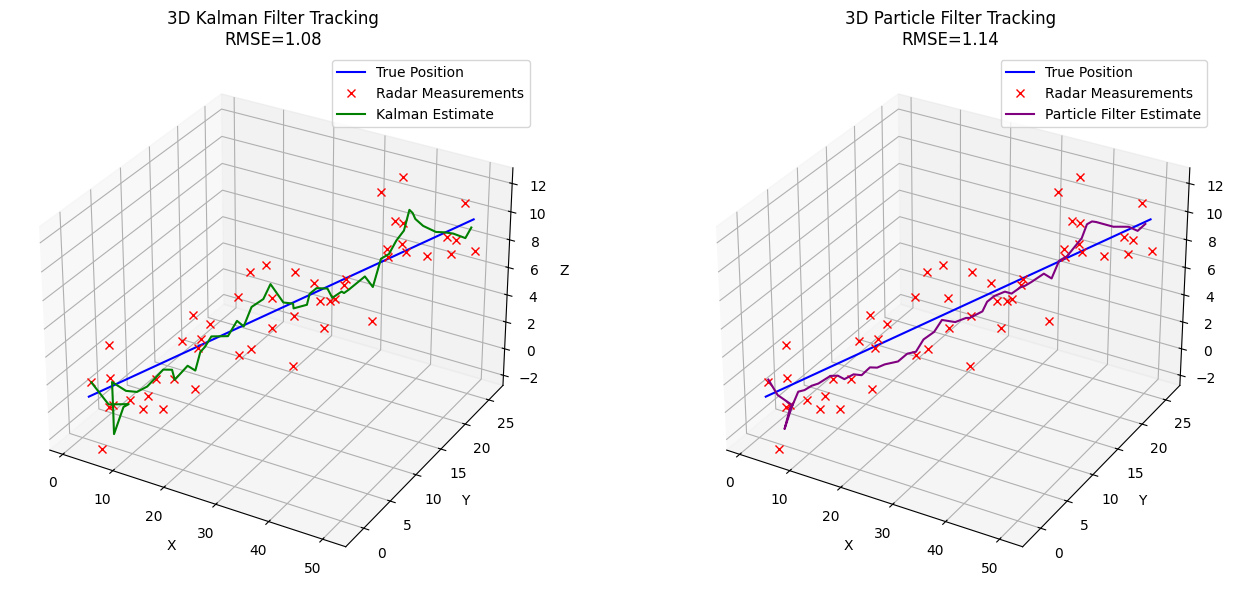


Figure 1: 3D Kalman Filter and Particle Filter Tracking

**B. Per-step Tracking Error**

Figure 2 presents the **Root Mean Squared Error (RMSE)** for both Kalman Filter and Particle Filter estimates over each time step. This plot provides a temporal view of how the accuracy of each filter evolves during the tracking process. Lower RMSE values indicate better tracking performance.

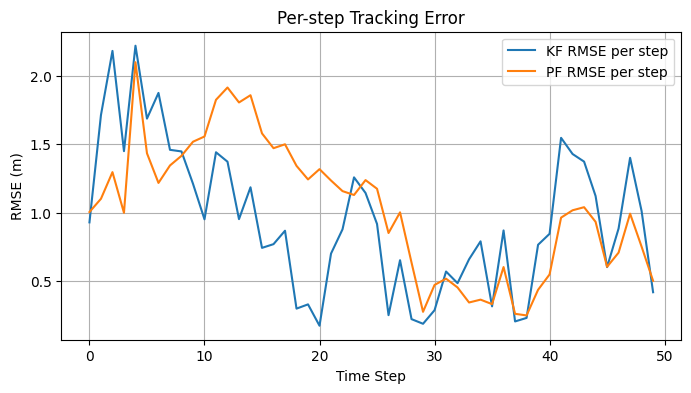
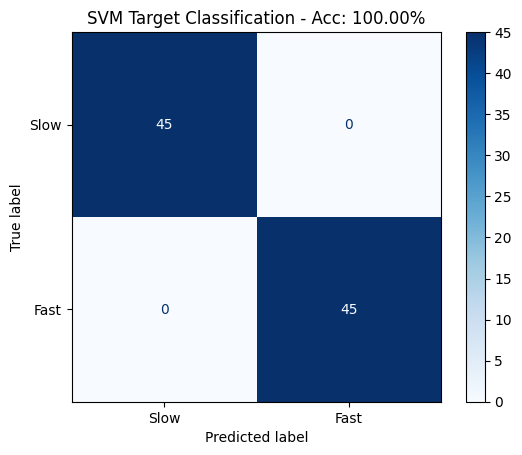


Figure 2: Per-step Tracking Error

**C. Classifier 1: Support Vector Machine (SVM)**

An **SVM** is trained on the scaled synthetic motion features. The implementation includes an option for GridSearchCV to find the optimal hyperparameters (C, gamma, kernel) for the SVM, enhancing its performance. The class\_weight='balanced' parameter is used to handle potential class imbalances in the dataset. The trained SVM's performance is evaluated using accuracy, a classification report (precision, recall, f1-score), and a confusion matrix

  
Figure 3: SVM Target Classification Confusion Matrix

**D. Classifier 2: Random Forest (Optional Comparison)**

As an optional comparison, a **Random Forest Classifier** is also implemented. Random Forests are ensemble learning methods capable of performing both classification and regression tasks. They build multiple decision trees during training and output the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. This model is also configured with class\_weight='balanced' and its performance is evaluated. Unlike SVMs, tree-based models like Random Forests do not strictly require feature scaling.

The feature importances from the Random Forest model are also extracted and visualized, providing insight into which motion features are most influential in determining target class.

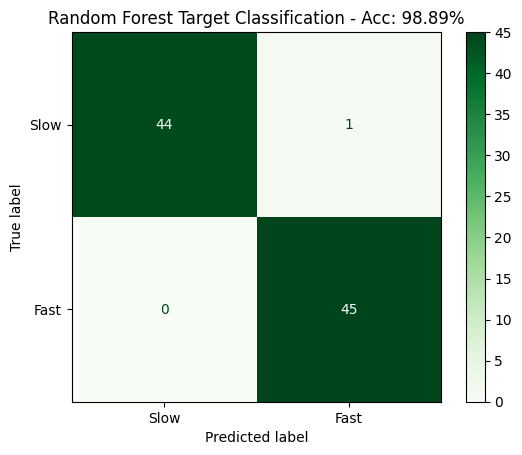


Figure 4: Random Forest Target Classification Confusion Matrix

**VI. APPLICATIONS**

The capabilities demonstrated by this project—accurate target tracking and motion-based classification—have broad applications across various domains:

* **Autonomous Vehicles (Self-Driving Cars, Drones):**
  + **Collision Avoidance:** Radars provide crucial distance and velocity information, enabling vehicles to detect obstacles and other vehicles, preventing accidents.
  + **Adaptive Cruise Control:** Maintaining a safe following distance from vehicles ahead by continuously tracking their speed and position.
  + **Traffic Monitoring:** Classifying vehicle types (cars, trucks, motorcycles) to optimize traffic flow and identify anomalies.
  + **Navigation in Adverse Weather:** Unlike cameras, radar performs well in fog, heavy rain, or snow, making it vital for reliable autonomous operation in challenging conditions.
* **Air Traffic Control (ATC):**
  + **Aircraft Tracking:** Continuously monitoring the position and velocity of aircraft to ensure safe separation and efficient routing.
  + **Conflict Detection:** Identifying potential collisions between aircraft and issuing alerts.
  + **Target Identification:** Differentiating between various types of aircraft (e.g., commercial jets, small planes, helicopters) based on their flight patterns and speeds.
* **Security and Surveillance:**
  + **Border Patrol and Perimeter Security:** Detecting and tracking intruders (humans, vehicles) across large areas, especially in low-visibility conditions.
  + **Critical Infrastructure Protection:** Monitoring sensitive sites like power plants or airports for unauthorized entry.
  + **Drone Detection:** Identifying and tracking drones in restricted airspace, distinguishing them from birds or other non-threats based on their unique flight characteristics.
* **Weather Forecasting:**
  + **Precipitation Monitoring:** Tracking the movement and intensity of rain, snow, and hail for accurate weather predictions and severe weather warnings.
  + **Storm Tracking:** Classifying storm types and predicting their paths and potential impact.
* **Robotics and Industrial Automation:**
  + **Human-Robot Collaboration:** Ensuring worker safety by tracking human movement around industrial robots.
  + **Automated Material Handling:** Guiding autonomous guided vehicles (AGVs) in warehouses to track and classify inventory items or other vehicles.
* **Sports Analytics:**
  + **Player Tracking:** Analyzing player movement patterns in sports (e.g., soccer, basketball) to derive performance metrics and tactical insights.
  + **Ball Tracking:** Precise tracking of ball trajectories in sports like baseball or golf for performance analysis.

**VII. CONCLUSION**

The "Real-Time Radar Target Tracking and Classification" project successfully integrates various signal processing and machine learning techniques to simulate a functional radar system. The implementation of both Kalman and Particle Filters provides robust solutions for diverse tracking challenges, from linear to non-linear target dynamics. The classification phase, utilizing SVM and Random Forest models on engineered motion features, demonstrates the capability to discern between different target types based purely on their movement patterns.

The visual analytics, including 3D trajectory plots and per-step RMSE curves, offer clear insights into the performance of the tracking algorithms, highlighting their strengths and limitations in handling noisy data. The high accuracy achieved by the classification models on synthetic data suggests the potential for real-world applications, provided the synthetic data adequately represents real-world complexities.

This project serves as an excellent educational and developmental platform for anyone interested in radar systems, state estimation, and the application of machine learning in sensor data analysis. Its modular design allows for future expansion, such as integrating real sensor data, exploring more advanced filtering techniques (e.g., Unscented Kalman Filter), or implementing deep learning models for classification.