

Project Proposal: Particle Filtering for Real-Time Drone Swarm Tracking (particle_filter)

Zev McManus-Mendelowitz (zem2109)

October 18, 2026

1 Motivation

I am going to join a drone interceptor company after I (hopefully) graduate this semester, so I want to use this project as a chance to get more familiar with an algorithm relevant to that domain.

I think particle filtering could be a good fit. At a high level, particle filtering algorithms try to estimate the position and motion of some object based on noisy sensor data. Traditional object tracking methods make a lot of assumptions which don't apply to drones, like normally distributed error on the measurements and linear motion. Particle filtering is sample-based and non-parametric, meaning it doesn't try to fit the "heatmap" of the object's possible states onto some specific distribution.

Instead, particle filtering relies on a loop of sampling lots of points (representing states with dimensions for position, velocity, rotation, etc...), estimating the likelihood that the object is in those states, re-sampling with higher density in the higher likelihood regions, and then recalculating the point-likelihoods as new input data comes in. At every timestep, this gives the probability distribution of the target object state as a cloud of discrete points which can fit any distribution. This robustness is very useful for tracking drones as they perform aggressive evasive maneuvers and radar measurements suffer from clutter and false returns.

The primary drawback of the particle filter algorithm is computational intensity since it requires processing hundreds of thousands of particles in real time. I'm hoping to use parallel functional programming to reduce this barrier.

2 The Particle Filter Algorithm

A particle filter maintains N particles, where each particle represents a hypothesis about the target drone's 3D position, velocity, and acceleration. As noisy radar measurements arrive at each discrete time step t , the particle cloud is updated through four distinct operations.

2.1 High-Level Algorithmic Steps

1. **State Propagation (Prediction):** Advance each particle's state forward in time using a 3D kinematic physics model, adding stochastic process noise to model unexpected target maneuvering. This can be done in parallel for each particle.
2. **Measurement Weighting:** Evaluate incoming radar observations against each particle's predicted state. Assign each particle an importance weight using a Gaussian likelihood function. This can also be completely parallelized.
3. **Weight Normalization:** Sum all particle weights and divide each individual weight by the total sum so the weights form a valid probability distribution (summing to 1.0). This can be optimized with parallel sums (tree reductions).

4. **Resampling:** Discard low-weight particles and replicate high-weight particles, resetting all particle weights to $1/N$ to focus computational resources on probable state regions. This can be optimized through parallel prefix scans and partitioning (more details below).

3 Resampling Strategies

Resampling represents the primary computational bottleneck because standard systematic resampling requires a global prefix sum scan across all N particles. There are more parallelizable approaches, but they give up some mathematical "optimality."

To explore this time/accuracy trade-off, I plan to implement and compare two distinct resampling strategies in Haskell:

1. **Global Systematic Resampling:** Builds a single global CDF across all N particles. This will be implemented with a global parallel prefix sum scan followed by gather lookups. While mathematically optimal in minimizing sample variance, it requires global thread synchronization.
2. **Local / Block-Based Resampling:** Partitions the particle array into local blocks (matching thread or GPU block boundaries). Each block normalizes and resamples locally without global communication, accompanied by periodic inter-block particle swaps. This achieves speedup by keeping each working block small enough to fit entirely inside L1 cache.

4 Haskell Parallel Architecture & Implementation Strategy

4.1 Outer Level: Swarm Task Parallelism (`Control.Monad.Par`)

When tracking a swarm of M independent target drones simultaneously, each drone's particle filter operates independently. I'll use `Control.Monad.Par` to spawn task threads across CPU cores using a `parMap` pattern since it gives deterministic scheduling without lazy evaluation memory leaks.

4.2 Inner Level: Particle Array Data Parallelism (REPA)

Inside each drone's filter, fine-grained array operations ($N = 10^4 \dots 10^6$ particles) are parallelized using REPA. REPA's delayed array fusion collapses prediction, weighting, and normalization passes into single-pass vector computations.

4.3 Optional Extension: GPU Acceleration (`Accelerate`)

If time permits after validating the REPA CPU baseline, I'll port the inner data-parallel array operations to `Accelerate` to compile and benchmark particle filters directly on GPU hardware via the CUDA/LLVM backend. I'll probably do this on Google Cloud since I don't own a GPU.

5 Data Sourcing

To keep the project focused strictly on parallel functional performance rather than physics simulation, input datasets will be pre-generated offline using Python's open-source defense tracking framework, `Stone Soup` (developed by DSTL).

Synthetic 3D target trajectories featuring evasive maneuvers (constant-velocity legs interspersed with coordinated turns) will be generated and exported to structured CSV files. Sensor observations will simulate 3D radar returns (converting Cartesian ground truth to range, azimuth, and elevation) with added Gaussian measurement noise ($\sigma = 5\text{--}15$ meters) and random

false-alarm clutter. The pre-baked telemetry files will be parsed at runtime in Haskell using the `cassava` CSV decoding library.