Target Tracking and Motion Planning with Particle Filters for a Multi-Agent Line of Sight Network

Collin Hudson
Aerospace Engineering Sciences
University of Colorado Boulder
collin.hudson@colorado.edu

***Abstract*—This work investigates the application of particle filtering techniques for a multi-agent path planning system with the goal of maintaining line of sight between a set of Uncrewed Aircraft Systems (UAS) and an unknown target to form a continuous communication network in a cluttered environment. In order to plan for the network, the target state is estimated using a particle filter, based on range-only measurements from each UAS. The motion-planning problem at each time step is then solved as an optimization problem to maximize the number of UAS that are in position to measure the target while maintaining network connectivity between all UAS agents and the ground station. The effectiveness of several variants of the SIR particle filter were also evaluated, based on their ability to estimate the target state over all time steps. Solutions to the problem were generated using the SCIP optimizer and demonstrated via simulation using the Julia programming language.**

I. Application and Context

This project seeks to address limitations faced by communication systems that rely on Line Of Sight (LOS) when operating in obstructed environments. For example, consider the case of a search-and-rescue team that uses radios to communicate with a mission coordinator at a base camp in a mountainous region. Fixed radio repeaters necessitate valuable time and resources from the rescue operation to manually set up, and can't adapt to unforeseen communication obstructions. Instead, the radio repeaters could be mounted to UAS agents that could form a network that ensures line of sight between its members. In addition, the problem of maintaining line of sight with some actor or agent in an environment has a variety of applications that the UAS network could help address. For this project, it is assumed that there is a fixed ground station that all UAS must have a path to connect to through the network, and another agent that the network wants to maintain line of sight, referred to generally in this project as the target.

In many cases, such as GPS-denied environments, the actual position of the target is unknown and must be estimated based on measurements taken by the UAS agents in the network. As a result, the estimator and motion planner work together to both infer the position of the ground agent based on the pseudorange measurements and plan for the motion of the UAS network to take the aforementioned measurements. The principal uncertainty to consider is that of the target position at the next time step so that the motion planner can provide positions for each of the sensing UAS agents to take the next measurement. In order to maintain a reasonable scope for the project, the target is assumed to have a constant but unknown velocity, reducing the future state uncertainty to the position of the target at the current time step and its velocity. Since the UAS agents can only provide range measurements, probability distributions of the target position are highly non-Gaussian. For example, if one UAS agent receives a range measurement of the target, the resulting position probability distribution would be a noisy ring around the agent, which would be poorly represented by a single normal distribution.

While this complexity could possibly be addressed using a Gaussian mixture, the distributions are also subject to several point-based constraints such as line of sight with the sensing agent, collision avoidance with obstacles, and an assumed maximum communication radius. As a result, distributions over areas could overlap with obstacles or in areas outside the view of the UAS agent, leading to poor target position estimates. Thus, the estimation portion of the problem naturally lends itself to approximate inference techniques, and intuitively the particle filter, which can immediately handle the point-based constraints simply based on the choice of particle reweighting function. The choice of particle filter is also non-trivial, as the all-or-nothing nature of the finite-range measurements in a cluttered environment could quickly lead to particle depletion depending on the choice of importance sampling function. In addition, naive approaches like the bootstrap particle filter may lead to worse target estimates by being unable to leverage information from previous time steps during sampling, compared to more complex approaches that can utilize domain-specific importance sampling functions.

II. Probabilistic Model

TABLE I: Probabilistic model variables

Variable	Definition
n_{uas}	Total number of UAS agents
$\mathbf{x}_{g_k}$	State of target at time step k, [position,velocity]^T
$\mathbf{x}_{a_{i,k}}$	Position of UAS agent i at time step k
$\mathbf{x}_k$	State vector at time step k, $[\mathbf{x}_{g_k}^T, \mathbf{x}_{a_{1,k}}^T, \ \dots \ , \mathbf{x}_{a_{n_{uas},k}}^T]^T$
$\mathrm{y}_{i,k}$	Measurement taken by UAS agent i at time step k
$\mathbf{y}_k$	Measurement vector at time step k, $[y_{1,k}, \ \dots \ , y_{n_{uas},k}]^T$
$\mathrm{v}_{i,k}$	Measurement Noise for UAS agent i at time step k

For this project, the target has no outside disturbances and can be described using a DBN with Markov properties. It is also assumed that the measurements taken by each UAS agent are independent with respect to each other agent as well as with time. The DBN below shows the relationship between the target and UAS agent $i \in \{1, 2, \ \dots \ , n_{uas}\}$ up to an arbitrary time step $k \in \{1, 2, \ \dots \ , k_{end}\}$.

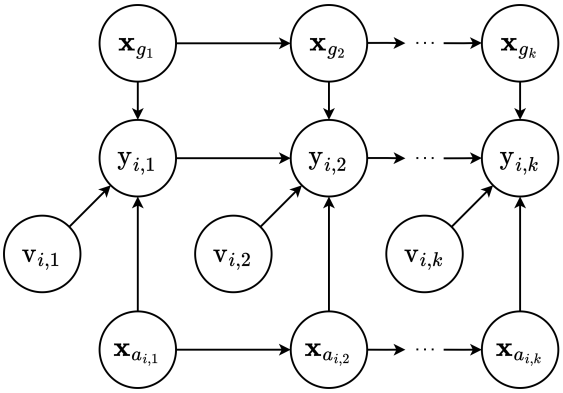


Fig. 1: Dynamic Bayesian Network for UAS agent i

The measurement model is given below, with measurement noise $v_{i,k} \sim \mathcal{N}(0, R_i)$ and distance between agents $r_{i,k} = \left\|\mathbf{x}_{g_k} - \mathbf{x}_{a_{i,k}}\right\|_2$. The model is intended to represent an agent receiving a signal from the target and calculating the distance between them from the pseudorange, with the noise v_{i_k} encapsulating any errors between the pseudorange and true range.

$$y_{i,k} = \begin{cases} r_{i,k} + v_{i,k} & \text{if } \ r_{i,k} \leq r_{\text{con}} \text{ and in LOS} \\ \text{inf} & \text{o.w.} \end{cases}$$

Because the measurement noise is assumed to be AWGN, the probability of receiving measurement $y_{i,k}$ given the state $\mathbf{x}_k$ could be modeled using a truncated Gaussian distribution with bounds at 0 and the maximum connection radius r_{con}. This will prove to be useful, as methods for sampling from truncated Gaussian distributions are well-known and readily available. Let Φ represent the CDF of a normal distribution. The measurement noise variance for each agent is assumed to be equal based on the assumption that all UAS agents are identical, and set to 0.01 based on simulation performance, $R_i = 0.01 \ \forall i \in \{1, ..., n_{uas}\}$. The maximum connection radius r_{con} is also an arbitrary parameter that merely scales how much of the environment each agent can search for the target in a single time step. As a rule of thumb, the connection radius is limited to one quarter of the width of the environment.

$$P(y_{i,k}|\mathbf{x}_k) \sim \frac{\mathcal{N}(r_{i,k}, \ R_i)}{\Phi\left(\frac{r_{con} - r_{i,k}}{R_i}\right) - \Phi\left(\frac{-r_{i,k}}{R_i}\right)} \mathbb{I}(0 \leq r_{i,k} \leq r_{\text{max}})$$

Because the measurements taken by each UAS agent at every time step are assumed to be independent, the joint probability of receiving the measurements from all UAS agents given the positions of all agents at time step k is equal to the product of the probability of receiving each agent's measurement.

$$P(\mathbf{y}_k|\mathbf{x}_k) = \prod_{i=1}^{n_{uas}} P(y_{i,k}|\mathbf{x}_k)$$

The state transitions are deterministic, with the target state following the discrete time-invariant linear system below.

$$\mathbf{x}_{g_{k+1}} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x}_{g_k}$$

It should be noted that the true state of the target is unknown, and must be inferred at each time step using the measurements $\mathbf{y}_k$. For the first two objectives, the UAS agent position variables $\mathbf{x}_{a_{i,k}}$ are known and serve as inputs to the model, as they are specified by the motion planner for all time steps. For the final objective, the UAS agent positions are no longer known, and must also be estimated before planning can occur.

III. Objectives

The project objectives are given below, as defined in the project proposal.

1) **Implement bootstrap particle filter**
   Given a set of 3 UAS agents with known positions for all times, estimate the position of a ground agent using the noisy range measurement model and constant-velocity target model with a bootstrap particle filter.
2) **Generate trajectories for UAS LOS network**
   Given a fixed ground station position and n_{uas} agents, generate optimal network trajectories to maximize information / minimize entropy that respect network connectivity and obstacle constraints. UAS agents share position knowledge.
3) **UAS network localization**
   UAS agents no longer have perfectly known positions, so their positions must be inferred from relative measurements of range from LOS connections to each other and to the ground station. The ground station position is still fixed and known a priori.

As an off-ramp for objective 2, an alternative objective to minimizing the expected entropy in the target position distribution is using another form of the particle filter that better suits the problem, and comparing performance with the bootstrap PF.

IV. Methods: Objective 1

The bootstrap particle filter was implemented using the ParticleFilters.jl package [1], which offers a simple framework for customizing the prediction, reweighting, and resampling steps of a particle filter, along with a simplified framework when using a bootstrap particle filter. The measurement model was defined with the Distributions.jl package, which supports sampling from truncated Gaussians.

A. Bootstrap particle filter

The bootstrap particle filter is a straightforward implementation of the particle filter which uses the transition probability $P(\mathbf{x}_{k+1}|\mathbf{x}_k)$ as the importance sampling function $q(\mathbf{x}^i_{k+1};\mathbf{x}^i_k,\mathbf{y}^i_{k+1})$. This results in a particle weight update prior to resampling in the following form.

$$\begin{aligned} w^i_{k+1} &\propto \frac{P(\mathbf{y}_{k+1}|\mathbf{x}^i_{k+1})P(\mathbf{x}^i_{k+1}|\mathbf{x}^i_k)}{q(\mathbf{x}^i_{k+1};\mathbf{x}^i_k,\mathbf{y}^i_{k+1})}w^i_k \\ &\propto P(\mathbf{y}_{k+1}|\mathbf{x}^i_{k+1})w^i_k \end{aligned}$$

In addition, the particle set is resampled after every measurement update, with weights normalized following the resampling step.

The implementation using ParticleFilters.jl is also fairly straightforward using the `BootstrapFilter()` function, which requires as input a particle prediction function and a particle reweighting function. The resampling and weight normalization steps are implemented within the filter object.

The prediction function is simply the particle multiplied by the state transition matrix, plus an artificial process noise vector drawn from a zero mean multivariate Gaussian distribution with small variances. While the real transitions are deterministic, the particle filter would often collapse to a single particle far from the actual target state, so a small amount of process noise was added. This also helps add robustness to the estimation algorithm, as the true target may not have constant velocities.

Algorithm 1 Bootstrap particle filter reweighting function

if x^i_{k+1} is outside bounds or in obstacle **then**
  return 0
else
  $w^i_{k+1} \leftarrow \mathcal{N}(0,R_v)|_{\Delta v}$
  for j = 1:n_{uas} **do**
    $r_{j,k} \leftarrow \left|\left|\mathbf{x}^i_k - \mathbf{x}_{a_{j,k}}\right|\right|_2$
    if $y^j_{k+1} = -1$ (No measurement received) **then**
      $w^i_{k+1} \leftarrow w^i_{k+1} * (r^j_k > r_{con})$
    else if $r^j_k \leq r_{con}$ **then**
      $w^i_{k+1} \leftarrow w^i_{k+1} *$ truncated$(\mathcal{N}(r_{j,k},R_j))|_{y^j_{k+1}}$
    else
      return 0
    end if
  end for
  return w^i_{k+1}
end if

The particle reweighting function returns a weight for particle i at time step $k+1$. The function takes as input the particle at time step k, and the predicted particle state at time step $k+1$, along with the measurements and UAS agent positions at step $k+1$. Within the particle weighting function, each particle is first weighted by the probability of drawing the difference in the change in position between time steps and the velocity of the particle at the last time step (Δv) from a zero mean Gaussian distribution. This is intended to weight particles according to how well the change in position matches the velocity, which should be constant.

V. Methods: Objective 2

A. Auxiliary particle filter

The auxiliary particle filter was implemented following the filter description and algorithm outline in [2], again using the ParticleFilters.jl package. This filter was chosen based on its interesting architecture and successful application to a 2D target detection and tracking problem in [3], with better performance than the bootstrap particle filter.

In the auxiliary particle filter, rather than resampling after the prediction and reweighting steps, a pre-selection step is performed before the prediction and reweighting steps. The goal is to select samples to propagate that are the most likely to become high-weight particles after the reweighting step.

To accomplish this, an auxiliary variable i is introduced which acts as an index for each particle. Then, auxiliary weights λ_i are calculated for each i proportional to the probability of receiving the measurement given the mean of the predicted state of the ith particle multiplied by the particle's weight. From the distribution of i weights, N_p indices are sampled to form the set of indices of particles to be propagated through the prediction and reweighting steps, where N_p is equal to the total number of particles used for filtering. The pre-selection function is given below, and takes as input the weighted particles from the step k, along with the measurements and UAS agent positions at time step $k+1$. The function returns a new set of weighted particles.

In order to combat particle depletion due to a surprising measurement, the algorithm outlined in [2] was modified such that if the auxiliary weights sum to very near zero, the auxiliary variable distribution is set to a uniform distribution over all particle indices. While this check was not included in the algorithm description, it completely addressed filter failures in challenging environments, and can be thought of as falling back to the bootstrap form for that step.

Algorithm 2 Auxiliary particle filter pre-selection function

for $i = 1 : N_p$ **do**
  $\lambda_i \leftarrow w^i_k * P(\mathbf{y}_{k+1}|A\mathbf{x}^i_k)$
end for
if $\sum_i^{N_p} \lambda_i \approx 0$ **then**
  for $i = 1 : N_p$ **do**
    $\lambda_i = \frac{1}{N_p}$
  end for
else
  Normalize weights $\lambda_i \ \forall i \in \{1,...,N_p\}$
end if
for $j = 1 : N_p$ **do**
  $\hat{i} \sim P(\{\hat{i} = i\}) = \lambda_i$
  Add $\left(\mathbf{x}^{\hat{i}}_k, w^{\hat{i}_k}\right)$ to weighted particle set
end for
return weighted particle set

The reweighting step is slightly modified to divide the probability of receiving a measurement given the particle state by the probability of receiving said measurement given the mean predicted state of the particle at the last time step.

Algorithm 3 Auxiliary particle filter reweighting function

if x^i_{k+1} is outside bounds or in obstacle **then**
  return 0
else
  $w^i_{k+1} \leftarrow \mathcal{N}(0, R_v)|_{\Delta v}$
  for j = 1:n_{uas} **do**
    $r_{j,k} \leftarrow \left|\left|\mathbf{x}^i_k - \mathbf{x}_{a_{j,k}}\right|\right|_2$
    $\mu_{j,k} \leftarrow \left|\left|A\mathbf{x}^i_{k-1} - \mathbf{x}_{a_{j,k}}\right|\right|_2$
    if $y^j_{k+1} = -1$ (No measurement received) **then**
      $w^i_{k+1} \leftarrow w^i_{k+1} * (r^j_k > r_{con})$
    else if $r^j_k \leq r_{con}$ **then**
      $w^i_{k+1} \leftarrow w^i_{k+1} * \frac{\text{truncated}(\mathcal{N}(r_{j,k}, R_j))|_{y^j_{k+1}}}{\text{truncated}(\mathcal{N}(\mu_{j,k}, R_j))|_{y^j_{k+1}}}$
    else
      return 0
    end if
  end for
  return w^i_{k+1}
end if

B. Optimization Problem Formulation

For clarity, the ground station and target are collectively referred to as ground agents, and all other agents are referred to as UAS agents. The optimization problem can be summarized as follows: **Given the known ground station and estimated target positions, find valid positions and connections for n_{uas} available UAS agents that define a line of sight network which connects to all ground agents.** The agents must not collide with any obstacle in the environment, which are modeled as convex polygons. The problem is presented as a Mixed-Integer Nonlinear Programming problem at every time step for a finite number of time steps. The state is represented as follows:

TABLE II: Optimization state variables

Variable	Definition
x_i	x coordinate of agent i
y_i	y coordinate of agent i
$c_{i,j}$	Binary variable for connection between agents i and j
$o_{h,i}$	Binary variable for obstacle half-space constraint h for agent i
$l_{h,i,j}$	Binary variable for obstacle half-space constraint h at midpoint between agents i and j
a_i	Binary variable for flow reception
$f_{i,j}$	Flow from agent i to agent j

Let the position of ground agent i at time step k be specified by $g(i,k)$. Let the index sets for ground agents, UAS agents, obstacles, and obstacle edges be represented by $\mathcal{G}$, $\mathcal{U}$, $\mathcal{O}$, and $\mathcal{O}_e$, respectively. In addition, let the total number of obstacles in the environment equal n_{obs}, and the sum of the number of edges of all obstacles in the environment equal n_e.

1) Connection Constraints: The connection of two agents is limited by some communication radius r_{con} as an analog to real-world limits on signal strength. This constraint is implemented using the Big M method such that the constraint is completely relaxed if two agents are not connected.
Let $M_c = 1.1((x_{max} - x_{min})^2 + (y_{max} - y_{min})^2)$, where the minimum and maximum values are the bounds of the environment. The constraint is defined as follows:

$$||\mathbf{p}_i - \mathbf{p}_j||^2 \leq r^2_{con} + M_c(1 - c_{i,j}) \quad \forall i,j \in \mathcal{G} \cup \mathcal{U} \tag{1}$$

Thus, if agent i and j are not connected, the inequality becomes trivial. Since x and y are constrained to be within the bounds of the environment, their difference can never exceed M_c. As a result of this constraint, the Mixed-Integer Program is quadratic.

2) Obstacle Avoidance Constraints: The MIP formulation for obstacle avoidance mentioned in [4] was integrated into the problem formulation, which also utilizes Big M relaxation methods. The following constraints are added to the formulation. Let $\{E_j\}$ equal the set of indices of half-spaces corresponding to obstacle j.

$$-\mathbf{h_j}^T\mathbf{p}_i \leq -k_j + M_j(o_{j,i}) \quad \forall i \in \mathcal{U}, j \in \mathcal{O}_e \tag{2}$$

$$\sum_{k \in E_j} o_{k,i} \leq |E_j| - 1 \quad \forall i \in \mathcal{U}, j \in \mathcal{O} \tag{3}$$

Constraint 3 ensures that at least one constraint 2 for obstacle j is enforced, such that agent i is outside of at least one edge of obstacle j.

3) Network Constraints: The following constraints are derived from the formulation introduced in [5], with some modifications to fit the application. The goal of the constraints is to solve for connections between nodes and nonnegative units of units of artificial "flow" in and out of each node in a network, where each node must consume one unit of flow unless it is the designated source node. Since each node needs to consume a unit of flow, each node must be connected to a node with a nonnegative out flow, and therefore must have a path to the source node.

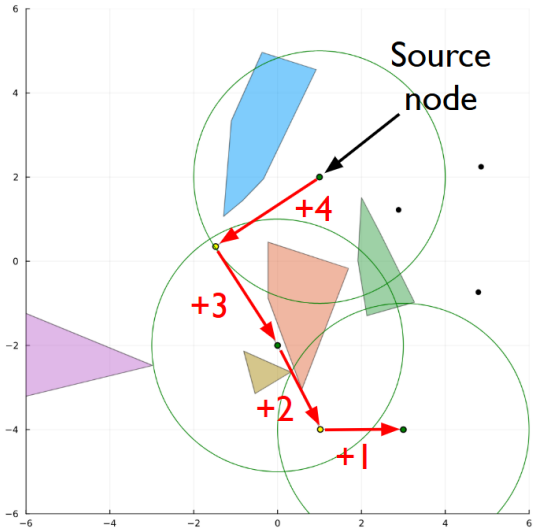


Fig. 2: Network flow example

In the formulation for the Minimum Connectivity Inference (MCI) problem discussed in [5], the subset of nodes, called a cluster, to be connected is known. For the LOS network application, the number of agents in the network is part of the state to be optimized, and is therefore unknown with respect to the MCI problem, requiring modifications to the MCI constraints. Summing over column i of the state variable f will give the total flow into agent i. Similarly, summing over row i gives the total flow out of agent i. For simplicity, the ground station is defined as the source node. The following constraints are added.

$$\sum_j \sum_i c_{i,j} \geq \sum_i (a_i) - 1, \tag{4}$$

$$\sum_j c_{i,j} \leq (n_{ga} + n_{uas}) a_i \quad \forall i \in \mathcal{G} \cup \mathcal{U}, \tag{5}$$

$$\sum_i c_{i,j} \leq (n_{ga} + n_{uas}) a_j \quad \forall j \in \mathcal{G} \cup \mathcal{U}, \tag{6}$$

$$\sum_j f_{i,j} \leq (n_{ga} + n_{uas}) a_i \quad \forall i \in \mathcal{G} \cup \mathcal{U}, \tag{7}$$

$$\sum_j f_{i,j} - \sum_k f_{k,i} = -a_i \quad \forall i \neq 1 \in \mathcal{U}, \tag{8}$$

$$f_{i,j} - f_{j,i} \leq (\sum_i (a_i) - 1) c_{i,j} \quad \forall i < j \in \mathcal{G} \cup \mathcal{U}, \tag{9}$$

$$f_{i,j} \geq 0 \quad \forall i = j \in \mathcal{G} \cup \mathcal{U} \tag{10}$$

Constraint 4 ensures that there are enough connections to connect all agents in the in the network. Constraints 5 and 6 ensure that agents must be in the network to have any connections. Constraint 7 ensures that agents must be in the network to have any flow out of them. Constraint 8 specifies that one unit of flow must be consumed when passing through an agent in the network, unless it is the source node (ground agent 1). Constraint 9 requires no net flow between unconnected agents, and that connected agents have a net flow less than the number of agents in the network. Constraint 10 defines flow as between different agents.

Because the network must act as a sensing network to estimate the target, and it is assumed that the UAS agent measurements are only received by the ground station if the agent is connected to the network, only the target is allowed to be outside of the network. This is crucial due to the fact that the estimated target position may change drastically between time steps when no connections have been made with the target yet. Thus, the following constraint is added.

$$a_i = 1 \quad \forall i \in \{(\mathcal{G} \cup \mathcal{U}) \setminus i_{target}\} \tag{11}$$

4) LOS Constraints: Constraints from the formulation introduced in [6] were included to ensure line of sight in the network. The nodes are considered in LOS if the midpoint $\mathbf{m_{i,p}}$ of the line between agents i and k belongs to both of the external half-spaces that contain each node. The following constraints are added:

$$-\mathbf{h_j}^T \mathbf{m_{i,p}} \leq -k_j + M_j(o_{j,i}) + M_j(l_{j,i,p,1}) \quad \forall i, p \in \mathcal{G} \cup \mathcal{U}, j \in \mathcal{O}_e \tag{12}$$

$$-\mathbf{h_j}^T \mathbf{m_{i,p}} \leq -k_j + M_j(o_{j,p}) + M_j(l_{j,i,p,1}) \quad \forall i, p \in \mathcal{G} \cup \mathcal{U}, j \in \mathcal{O}_e \tag{13}$$

$$c_{i,p} \leq 1 - l_{j,i,p,1} \quad \forall i, p \in \mathcal{G} \cup \mathcal{U}, j \in \mathcal{O} \tag{14}$$

Constraint 14 ensures that if agents i and p are connected, the LOS constraint must not be relaxed.

5) Time Step Constraints: In order for the solution paths to be viable, UAS agents must be able to reach their specified position at the next time step. A radius constraint is added for each UAS agent, with the radius corresponding to an assumed maximum UAS velocity.

$$||\mathbf{p}_{i,k} - \mathbf{p}_{i,(k-1)}||^2 \leq r_{uas}^2 \quad \forall i \in \mathcal{U}, \forall k > 1 \tag{15}$$

6) Objective Function: The estimation of the target state is reliant on range observations from the UAS agents, with more measurements resulting in a better estimate of the target state. Following from the problem definition, the number of measurements received at time step k is equal to the number of UAS agents within range and line of sight of the target at that time step. Since the constraints for connecting agents in the network are identical to the constraints for receiving a measurement, maximizing the number of network connections to the target at time step k is equivalent to maximizing the number of measurements of the state of the target at step k.

In reality, the true target state is uncertain, so the optimizer is not guaranteed to find the network configuration that maximizes the number of measurements of the target, but can only maximize the number of measurements of the most likely target position provided by the estimator. The objective function can be described as the summation of agents connected the the target as defined by the optimization variable c.

$$\begin{aligned} &\underset{\mathbf{x}}{\text{Maximize}} \sum_{j \in \mathcal{U}} c_{i_{target},j,k} \\ &\text{Subject to:} \\ &\text{Constraints 1-15} \end{aligned}$$

7) Solver: To solve the optimization problem at each time step, the problem was formulated using the JuMP package for the Julia programming language using the off-the-shelf optimizer SCIP [7]. The SCIP optimizer was chosen for its ability to handle Mixed-Integer Problems with quadratic constraints.

C. Motion planning and simulation

In order to evaluate both the estimator and optimizer, the following simulation function was implemented in the Julia programming language.

Algorithm 4 Simulation loop

$b \leftarrow$ uniform belief $\mathcal{U}$
$\mathbf{x}_{est,0} \sim b$
for $k = 1 : k_{end}$ **do**
  $\mathbf{x}_{g,k} \leftarrow A\mathbf{x}_{g,k-1}$
  Get agent states from optimizer with target = $A\mathbf{x}_{est,k-1}$
  for $i = 1 : n_{uas}$ **do**
    $r_k^i \leftarrow$ distance between $\mathbf{x}_{g,k}$ and agent i
    $y_k^i \sim \mathcal{N}(r_k^i, R_i)$
    if $y_k^i > r_{con}$ OR $\mathbf{x}_{g,k}$ out of LOS of agent i **then**
      $y_k^i \leftarrow -1$
    end if
  end for
  $b \leftarrow$ filter estimate given prior b, $\mathbf{y}_k$, and agent states
  $\mathbf{x}_{est,k} \leftarrow$ mode(b)
end for

VI. Results: Objective 1

In order to test the implementation of the bootstrap particle filter, a simple environment with fixed UAS agent positions was simulated using the loop described in the previous section with 10000 particles. As can be seen, the particle distributions match intuitively to the sensing radius intersection points, and the mode of the particle distribution follows the target closely for most time steps.

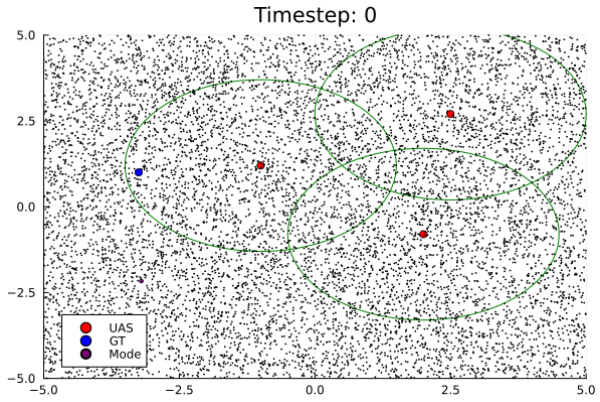


Fig. 3: Uniform initial belief

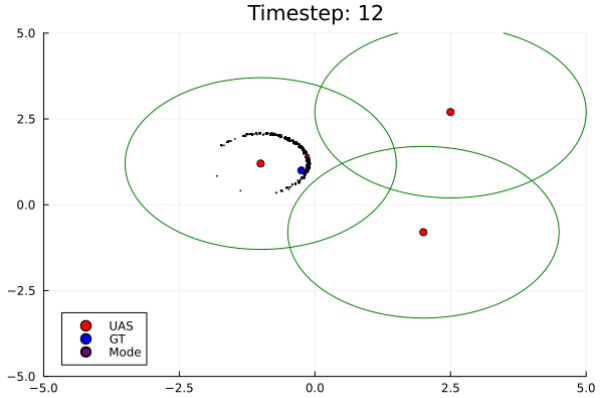


Fig. 4: Ring of belief due to single range measurement

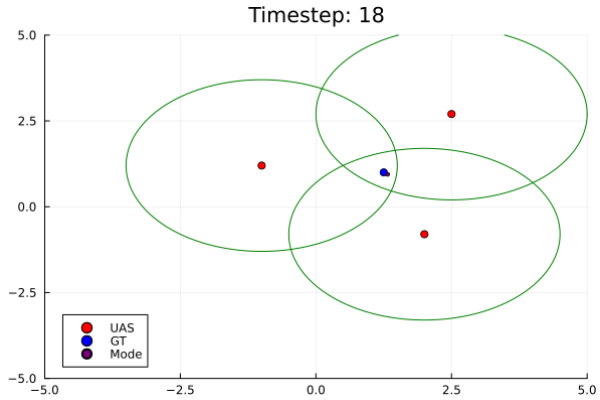


Fig. 5: Fully measured target

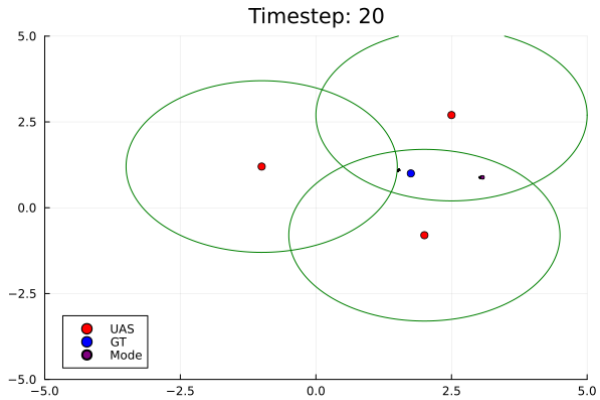


Fig. 6: Mode jumps to second intersection point

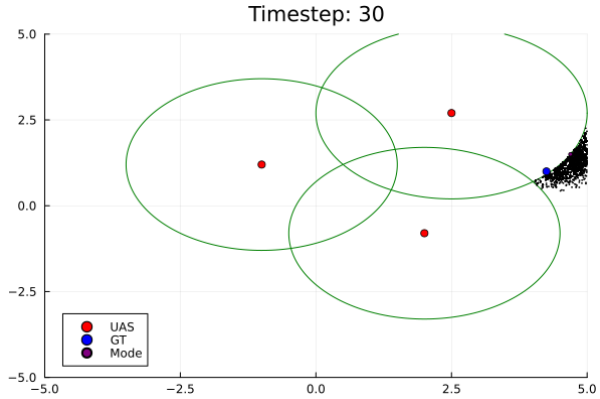


Fig. 7: Belief spreads out due to no measurements

However, the drawbacks to the bootstrap particle filter are also apparent, such as its inability to increase the weight of particles close to the true target due to a reduction in measurements.

VII. Results: Objective 2

The estimation algorithms were successfully integrated into the motion planner, and simulated in a cluttered environment. For the bootstrap filter, 25000 particles were used whereas only 500 particles were used with the auxiliary particle filter.

The auxiliary filter performed much better with fewer particles, with less weight collapses and thus less time steps with forced uniform index weights. The generated trajectories and target state estimates were generated using the same environment, target initial state, and ground station location.

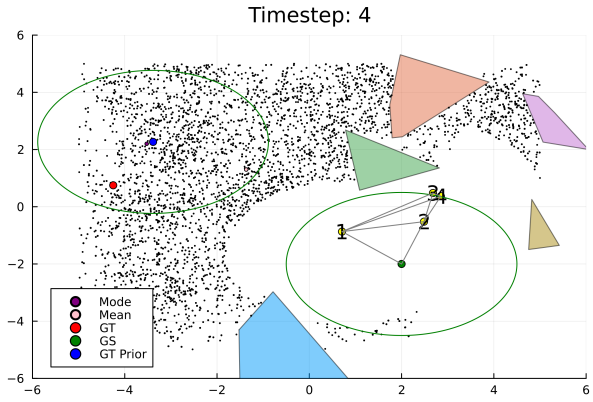


Fig. 8: Bootstrap particle filter

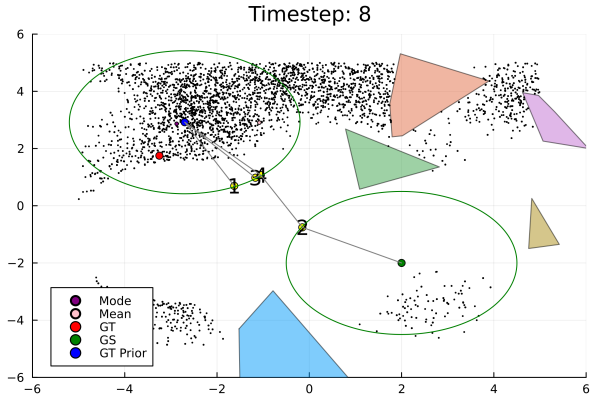


Fig. 9: Bootstrap: Effective search

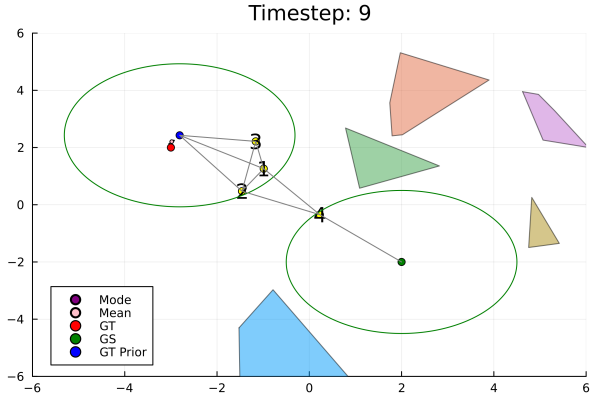


Fig. 10: Bootstrap: Target found

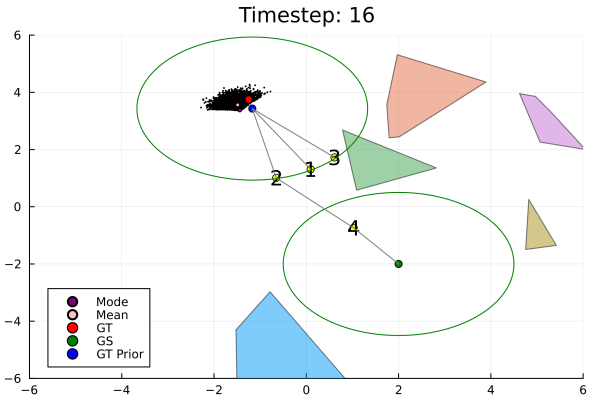


Fig. 11: Bootstrap: Moving out of range

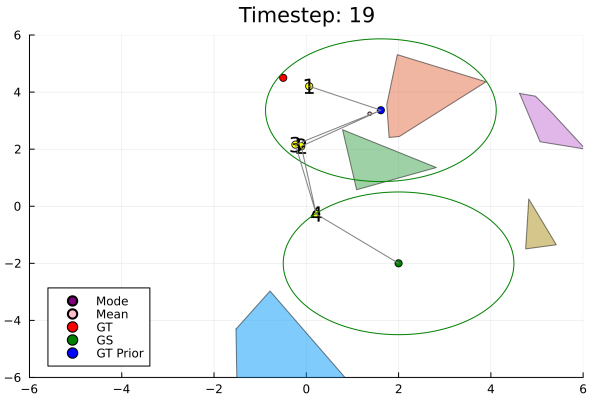


Fig. 12: Bootstrap: Estimate jump

The bootstrap filter is capable of finding and tracking the target, but is sensitive to momentary drops in measurements due to the UAS agents moving slightly out of range. In addition, the filter only cares about the transition from the last state, which can lead to large jumps in the target state estimate as seen in time step 19.
Using the auxiliary particle filter,

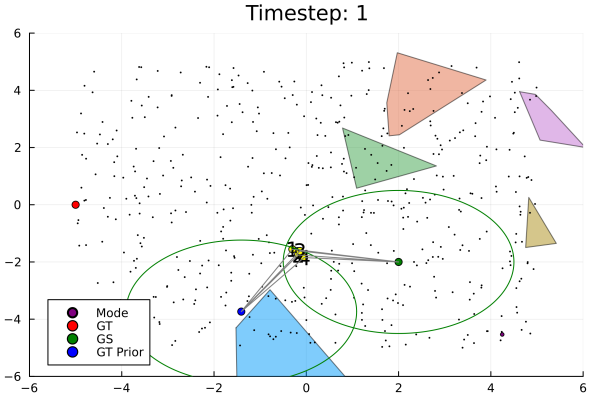


Fig. 13: Auxiliary particle filter

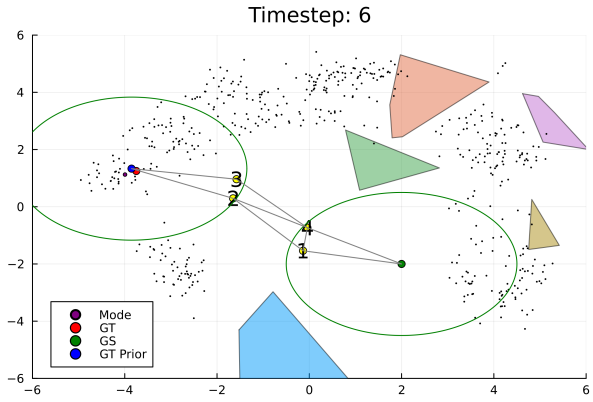


Fig. 14: Auxiliary: Effective search

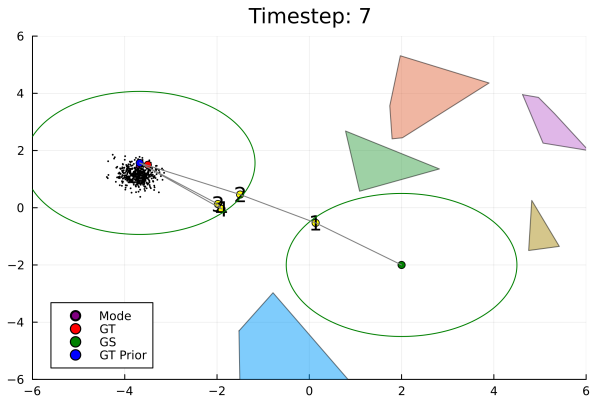


Fig. 15: Auxiliary: Target found

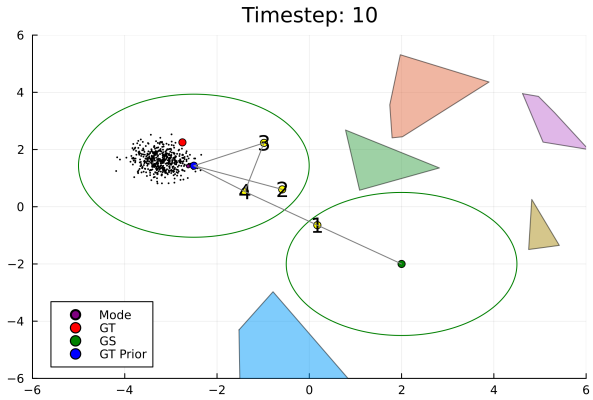


Fig. 16: Auxiliary: index weight collapse, particles spread out slowly

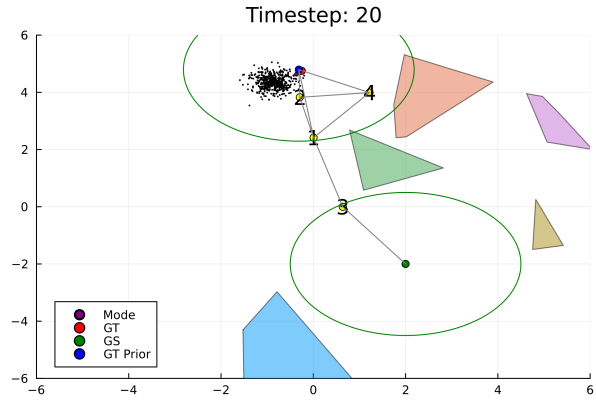
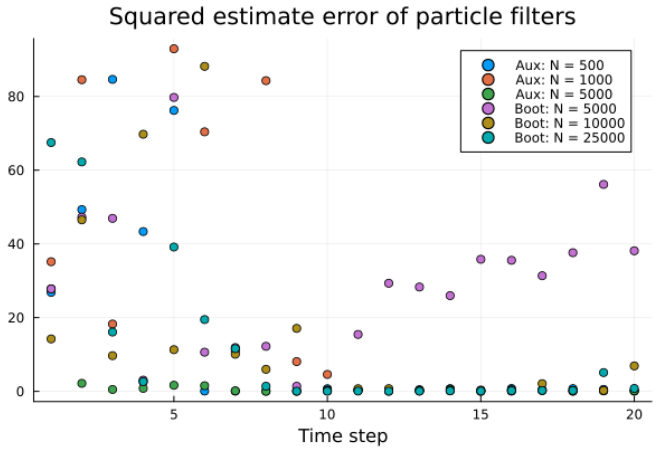
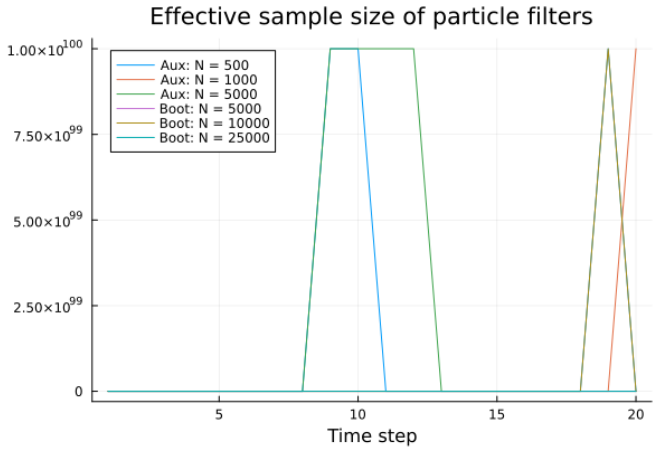


Fig. 17: Auxiliary: Estimate remains close to target

The sum of the squared target estimation error, equal to the distance between the estimated and true target positions squared, are plotted below with varied numbers of particles.



The effective sample size for each filter is shown below as well.



VIII. Discussion and Conclusions

The multi-agent LOS network problem for ensuring connectivity between a set of agents in a cluttered environment was

successfully solved using a Mixed-Integer Program approach and the SCIP optimizer for constraint integer programming. While valid solutions were demonstrated, further work could investigate the actual optimality of the generated paths with respect to the reduction in uncertainty of the target state. As shown by both the trajectory examples and estimate errors, the auxiliary particle filter enabled a significant reduction in the number particles needed to converge to the same amount of error as the bootstrap particle filter. However, the effective sample size does show that the auxiliary particle filter can spike periodically. with more particles leading to more spikes. It should also be noted that the bootstrap particle filter with 10000 particles also had a spike in effective sample size. Overall, I think the auxiliary particle filter was a useful addition to the project, and opens the door for distributed algorithms thanks to its reduction in necessary particles.

While the optimization formulation was able to successfully generate UAS agent trajectories that was capable of finding and tracking an unknown target, future work should be done investigating the feasibility of the generated paths. For example, the formulation considers a very simplistic disk model for agent movement, rather than the slightly more applicable Dubins vehicle models that could provide limits to turn rate and factor in vehicle orientation. In addition, a time step constraint that ensures the movement of an agent between time steps maintains line of sight with the past and present positions would further ensure there are no obstacle collisions between time steps. In terms of a path planner, I believe my solution could succeed, though it does rely on perfect knowledge of obstacle geometries, which is a fairly large assumption in uncertain environments.

Unfortunately, I did not have enough time to tackle the third objective, though I did have some ideas on how to tackle the problem. Since the optimization and path plans are sent from the ground station, the UAS agents could each run a particle filter and share measurements with their neighbors to estimate the position of the target and every UAS agent in the network. Before the network position is sent to the ground station, each UAS agent generates a belief about the position of all other agents, and shares it with their neighbors. Then, a series of consensus rounds could occur until the network converges on an estimate of their current position, which is then sent to the ground station for motion planning for the next step.

References

[1] Z. Sunberg, *Particlefilters.jl*, version v0.5.3, 2021. [Online]. Available: https : / / github . com / JuliaPOMDP / ParticleFilters.jl/tree/master.

[2] M. G. Bruno, "Regularized particle filters," in *Sequential Monte Carlo Methods for Nonlinear Discrete-Time Filtering*. Cham: Springer International Publishing, 2013, pp. 49–50, ISBN: 978-3-031-02535-8. DOI: 10.1007/978-3-031-02535-8_10. [Online]. Available: https://doi.org/10.1007/978-3-031-02535-8_10.

[3] M. G. Bruno, "Bayesian methods for multiaspect target tracking in image sequences," *IEEE Transactions on Signal Processing*, vol. 52, no. 7, pp. 1848–1861, 2004. DOI: 10.1109/TSP.2004.828903.

[4] D. Ioan, I. Prodan, S. Olaru, F. Stoican, and S.-I. Niculescu, "Mixed-integer programming in motion planning," *Annual Reviews in Control*, Nov. 2020. DOI: 10.1016/j.arcontrol.2020.10.008. [Online]. Available: https://centralesupelec.hal.science/hal-03108529.

[5] M. A. Dar, A. Fischer, J. Martinovic, and G. S. and, "An improved flow-based formulation and reduction principles for the minimum connectivity inference problem," *Optimization*, vol. 68, no. 10, pp. 1963–1983, 2019. DOI: 10.1080/02331934.2018.1465944.

[6] A. Caregnato-Neto, M. R. O. A. Maximo, and R. J. M. Afonso, "A novel line of sight constraint for mixed-integer programming models with applications to multi-agent motion planning," in *2023 European Control Conference (ECC)*, 2023, pp. 1–6. DOI: 10.23919/ECC57647.2023.10178275.

[7] S. Bolusani, M. Besançon, K. Bestuzheva, *et al.*, "The SCIP Optimization Suite 9.0," Optimization Online, Technical Report, Feb. 2024. [Online]. Available: https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/.

IX. Appendix

main.jl

```
# ASEN 5519 Final Project
# Collin Hudson 5/06/2025
using Plots
using Reel
using Printf
using Distributions
using Zygote
using LazySets
using JuMP, SCIP
using TickTock
using LinearAlgebra
using ParticleFilters
using Distances: Euclidean, colwise
ENV["TICKTOCK_MESSAGES"] = false
if isdefined(@__MODULE__, :workspace)
    workspace isa Module || error("workspace is present and it is not a Module")
else
    include("workspace.jl")
end
using .workspace

# Workspace parameters (xlim, ylim, number of obstacles, seed)
    # seed 2300 = squeeze
    # 353 broken somehow
    xTrue = [[-5.0;0;0.25;0.25]]
    gs = [2;-2]
    endK = 20
    e = env([-5,5],[-5,5],5)
    while !inLOS(xTrue[1][1],xTrue[1][2],xTrue[1][1] + endK*xTrue[1][3],xTrue[1][2] + endK*xTrue[1][4],e.obs) || ...
        inObs(gs[1],gs[2],e.obs)
        global e = env([-5,5],[-5,5],5)
    end

# Problem parameters
    Nga = 2
    maxUAS = 4
    Ntot = Nga+maxUAS
    rCon = 2.5
    rMin = max(0.1,rCon - 3)
    rUAS = 2
    vMax = 2.5
    samples = 0.5
    Mcon = 1.1*((e.xlim[2] - e.xlim[1])^2 + (e.ylim[2] - e.ylim[1])^2)
    R = 0.01.*I(maxUAS)
    A =  [1 0 1 0;0 1 0 1; 0 0 1 0; 0 0 0 1]
    wiggleCovBoot = [1 0 0 0;0 1 0 0;0 0 0.001 0; 0 0 0 0.001]
    wiggleCovAux = [0.05 0 0 0;0 0.05 0 0;0 0 0.001 0; 0 0 0 0.001]
    boot = true
    if boot
        Nparticles = 25000
    else
        Nparticles = 5000
    end
    xPath = []
    yPath = []
    xcPath = []
    cPath = []
    xEst = []
    plots = []
    NeffBoot = []
    NeffAux = []
    err = []
    Xmin = [e.xlim[1];e.ylim[1];0;0]
    Xmax = [e.xlim[2];e.ylim[2];sqrt(vMax)/4;sqrt(vMax)/4]
#

# Bootstrap Particle filter
    function dynPrediction(x,u,rng)
        # dynamics model for a constant velocity
        # input x = state at timestep k(column vector)
        # input u = UAS/sensor positions [unused]
        # input rng = rng seed (required by package)[unused]
        # returns x' = predicted state at timestep k + 1
        # f(x) = [1 0 1 0;0 1 0 1; 0 0 1 0; 0 0 0 1][xpos;ypos;xvel;yvel]
        return A*x + rand(MvNormal(wiggleCovBoot))
    end
```

```
    function obsWeight(xold,u,x,y)
        # weighting function given range only measurements from N sensors
        # input xold = state at timestep k(4x1 vector) [unused]
        # input u = UAS/sensor positions (Nx2 matrix)
        # input x = state at timestep k+1(4x1 vector)
        # input y = received measurement at timestep k+1 (Nx1 vector)
        # returns w = weight for particle with state x given recieved measurements y

        # Independent sensors (UAS) so probability of received measurement vector is
        #   product of probabilies of each sensor receiving measurement y[i]
        # w = 1
        if any(x .> Xmax) || any(x .< Xmin) || inObs(x[1],x[2],e.obs)
            return 0 # reject out-of-bounds particles
        end
        w = pdf(MvNormal([0;0],[0.05 0;0 0.05]),(x[1:2] - xold[1:2]) - xold[3:4]) #velocity error weight
        for j = axes(u,2)
            r = Euclidean()(x[1:2],u[:,j])
            measTrunc = truncated(Normal(r,R[j,j]); lower=0, upper=rCon)  #Gaussian truncated to the interval [l, u]
            if y[j] == -1 #check for NOT receiving a measurement (i.e. target is outside communication range)
                w *= (r > rCon)||(!inLOS(x[1],x[2],u[1,j],u[2,j],e.obs)) #reject particles that are inside the ...
                    measurement range and in LOS of sensor j
            elseif r <= rCon
                w *= pdf(measTrunc,y[j])
            else
                return 0 # reject particles outside sensor range when valid measurement received
            end
        end
        return w
    end

    function updateBoot(up::BasicParticleFilter, b::ParticleCollection, a, o)
        pm = up._particle_memory
        wm = up._weight_memory
        resize!(pm, n_particles(b))
        resize!(wm, n_particles(b))
        predict!(pm, up.predict_model, b, a, o, up.rng)
        reweight!(wm, up.reweight_model, b, a, pm, o, up.rng)
        push!(NeffBoot,sum(wm)^2/sum(wm.^2))
        return resample(up.resampler,
                        WeightedParticleBelief(pm, wm, sum(wm), nothing),
                        up.predict_model,
                        up.reweight_model,
                        b, a, o,
                        up.rng)
    end
#

# Auxilary PF functions

    function dynPredictionAux(x,u,rng)
        # dynamics model for a constant velocity
        # input x = state at timestep k(column vector)
        # input u = UAS/sensor positions [unused]
        # input rng = rng seed (required by package)[unused]
        # returns x' = predicted state at timestep k + 1
        # f(x) = [1 0 1 0;0 1 0 1; 0 0 1 0; 0 0 0 1][xpos;ypos;xvel;yvel]
        return A*x + rand(MvNormal(wiggleCovAux))
    end


    function resampleAux(b, u, y, rng)
        ows = zeros(Nparticles)
        map!(pj->obsWeight(pj,u,A*pj,y), ows, particles(b))
        ows = weights(b).*ows
        totW = sum(ows)
        if totW == 0 || isnan(totW) || isinf(totW)
            print("Auxiliary depletion (all weights zero)\n")
            ows = ones(Nparticles)./Nparticles
        end
        ows = ows./sum(ows) # Normalize auxiliary variable particle weights

        idx = [ParticleFilters.rand(WeightedParticleBelief(1:Nparticles,ows)) for j in 1:Nparticles]
        ws = [weight(b,i) for i in idx]
        return WeightedParticleBelief([particle(b,i) for i in idx],ws./(sum(ws)),sum(ws))
    end

    function obsWeightAux(xold,u,x,y)
        # weighting function given range only measurements from N sensors
        # input xold = state at timestep k(4x1 vector) [unused]
        # input u = UAS/sensor positions (Nx2 matrix)
        # input x = state at timestep k+1(4x1 vector)
        # input y = received measurement at timestep k+1 (Nx1 vector)
```

```
        # returns w = weight for particle with state x given recieved measurements y

        # Independent sensors (UAS) so probability of received measurement vector is
        #   product of probabilies of each sensor receiving measurement y[i]
        # w = 1
        # reject out-of-bounds particles
        if any(x .> Xmax) || any(x .< Xmin) || inObs(x[1],x[2],e.obs)
            return 0
        end
        w = pdf(MvNormal([0;0],[0.05 0;0 0.05]),(x[1:2] - xold[1:2]) - xold[3:4]) #velocity error weight
        for j = axes(u,2)
            r = Euclidean()(x[1:2],u[:,j])
            measTrunc = truncated(Normal(r,R[j,j]); lower=0, upper=rCon)  #Gaussian truncated to the interval [l, u]
            measTruncMean = truncated(Normal(Euclidean()((A*xold)[1:2],u[:,j]),R[j,j]); lower=0, upper=rCon)  ...
                #Gaussian truncated to the interval [l, u]
            if y[j] == -1 #check for NOT receiving a measurement (i.e. target is outside communication range)
                w *= (r > rCon) #reject particles that are inside the measurement range of sensor j
            elseif (r <= rCon) && (pdf(measTruncMean,y[j]) > 0)
                w *= pdf(measTrunc,y[j])/pdf(measTruncMean,y[j])
            else
                return 0 # reject particles outside sensor range when valid measurement received
            end
        end
        return w
    end

    function updateAux(up::BasicParticleFilter, b::WeightedParticleBelief, a, o)
        pm = up._particle_memory
        wm = up._weight_memory
        resize!(pm, n_particles(b))
        resize!(wm, n_particles(b))
        bnew = resampleAux(b, a, o, up.rng)
        predict!(pm, up.predict_model, bnew, a, up.rng)
        reweight!(wm, up.reweight_model, bnew, a, pm, o, up.rng)
        push!(NeffAux,sum(wm)^2/sum((wm./sum(wm)).^2))
        return WeightedParticleBelief(pm, wm, sum(wm), nothing)
    end

#

# Optimizer function
    function findStep(e, xLast, yLast, xcLast, k)
        model = Model(SCIP.Optimizer)
        set_silent(model)
        set_time_limit_sec(model,45)
        # Variables and workspace boundary constraints
            @variable(model, e.xlim[1] <= x[i=1:Ntot] <= e.xlim[2]) #Agent positions x
            @variable(model, e.ylim[1] <= y[i=1:Ntot] <= e.ylim[2]) #Agent positions y
            @variable(model, c[i=1:Ntot,j=1:Ntot], Bin) #Network edge choice (which nodes are connected)
            @variable(model, o[i=1:e.Nverts,j=1:Ntot],Bin) #Slack variables for obstacle avoidance
            @variable(model, l[r=1:e.Nverts,i=1:Ntot,j=1:Ntot,k=1:length(samples)],Bin) #Slack variables for LOS
            @variable(model, xc[i=1:Ntot],Bin) #Network cluster (active connection set)
            @variable(model, f[i=1:Ntot,j=1:Ntot] >= 0,Int) #Network flow
            @objective(model,Max, sum(c[2,:])) #Maximize connections to estimated target location
        #

        # Ground agent constraints
            for j in 1:Nga
                fix(x[j], xLast[j];force=true) #Ground agents must be at given x
                fix(y[j], yLast[j];force=true) #Ground agents must be at given y
                # fix(xc[j], 1;force=true) #Ground agents must be in network cluster
            end
            fix(xc[1], 1;force=true) #Ground station must be in network cluster

            for j in 3:Ntot
                fix(xc[j], 1;force=true) #all UAS must be in network cluster
            end
        #

        # Timestep constraints
            if k != 1
                for i in (Nga+1):Ntot
                    @constraint(model, (x[i] - xLast[i])^2 + (y[i] - yLast[i])^2 <= rUAS^2)
                    # fix(xc[i], xcLast[i];force=true)
                end
                # Add LOS constraint by adding columns? to o for each last position uas
            end
        #

        # Network constraints
            @constraint(model, sum(c*ones(Ntot,1)) >= (sum(xc)-1)) #Network must have at least (number of cluster ...
                agents - 1) edges
```

```
    for i in 1:Ntot
        @constraint(model, sum(c[i,:]) <= Ntot*xc[i]) #Only agents in cluster can be connected
        @constraint(model, sum(c[:,i]) <= Ntot*xc[i]) #Only agents in cluster can be connected
        @constraint(model, sum(f[i,:]) <= Ntot*xc[i]) #Only agents in cluster can have flow
        # @constraint(model, sum(f[:,i]) <= Ntot*xc[i]) #Only agents in cluster can have flow
        if i != 1
            @constraint(model,(sum(f[i,:]) - sum(f[:,i])) == -1*xc[i]) #Non-source cluster agents must ...
                consume one unit of flow
        end
        for j in 1:Ntot
            if i < j
                @constraint(model, (f[i,j] + f[j,i]) <= ((sum(xc) - 1)*c[i,j]))
            elseif i == j
                fix(f[i,j], 0;force=true) #Flow must be between two different agents
            end
        end
    end
#

# Connection radius constraint
    for i in 1:Ntot
        for j in 1:Ntot
            if i < j
                @constraint(model, (x[i] - x[j])^2 + (y[i] - y[j])^2 <= (rCon^2 + Mcon*(1-c[i,j]) + ...
                    Mcon*(1-xc[j]))) #pairs of active, connected agents must be within rCon of each other
                @constraint(model, (x[i] - x[j])^2 + (y[i] - y[j])^2 >= xc[j]*rMin^2) #pairs of active agents ...
                    must have at least rMin distance between each other
            else
                fix(c[i,j], 0;force=true) #do not consider order of pairs, i.e. consider (agent 1,agent 2) ...
                    and disregard (agent 2, agent 1) connections
            end
        end
    end
#

# Obstacle avoidance and LOS constraints
    row = 1
    obV = 1
    for ob in e.obs
        for con in constraints_list(ob)
            for j in 1:Ntot
                @constraint(model,-con.a'*[x[j];y[j]] <= (-con.b + e.Mobs[row]*o[row,j]))
            end
            for i in 1:Ntot
                for j in 1:Ntot
                    if i < j
                        for k in eachindex(samples)
                            si = (1-samples[k])*[x[i];y[i]] + samples[k]*[x[j];y[j]]
                            @constraint(model, -con.a'*si <= (-con.b + e.Mobs[row]*o[row,i] + ...
                                e.Mobs[row]*l[row,i,j,k]))
                            @constraint(model, -con.a'*si <= (-con.b + e.Mobs[row]*o[row,j] + ...
                                e.Mobs[row]*l[row,i,j,k]))
                        end
                        @constraint(model, c[i,j] <= sum(1 .- l[row,i,j,1:length(samples)])) #At least one ...
                            sample point must be valid if connected
                    end
                end
            end
            row += 1
        end
        for j in 1:Ntot
            @constraint(model,sum(o[obV:(obV+length(ob.vertices)-1),j]) <= (length(ob.vertices)-1)) #At least ...
                one constraint must be active per obstacle
        end
        obV += length(ob.vertices)
    end
#

optimize!(model)
if !is_solved_and_feasible(model)
    print("Infeasible workspace at k = "*string(k))
    # p = plot(size = (400,400))
    # for j in eachindex(e.obs)
    #     plot!(p,e.obs[j])
    # end
    # xlims!(p,e.xlim[1]-1,e.xlim[2]+1)
    # ylims!(p,e.ylim[1]-1,e.ylim[2]+1)
    # for j in 1:Ntot
    #     if j < Nga+1
    #         scatter!([xLast[j]],[yLast[j]],mc=:green,label=nothing)
    #         plot!(xLast[j] .+ rCon*cos.(range(0,2*  ,500)),yLast[j] .+ ...
        rCon*sin.(range(0,2*  ,500)),linecolor=:green,label=nothing)
```

```
            #     else
            #         scatter!([xLast[j]],[yLast[j]],mc=:black,label=nothing)
            #         plot!(xLast[j] .+ rUAS*cos.(range(0,2*  ,500)),yLast[j] .+ ...
                rUAS*sin.(range(0,2*  ,500)),linecolor=:gray,label=nothing)
            #     end
            # end
            # title!(p,"Infeasible workspace")
            # display(p);
        end
        if is_solved_and_feasible(model)
            return [value.(x),value.(y),ceil.(value.(xc)),ceil.(value.(c))]
        else
            return [xLast,yLast,xcLast,zeros(Ntot,Ntot)]
        end
    end
#

#  Helper functions
    function initBelief(Xmin,Xmax,Nparticles,boot)
        len = size(Xmin,1)
        p0 = zeros(Float64,len,Nparticles)
        for j = 1:Nparticles
            temp = rand(len)
            p0[:,j] = Xmin.*temp + Xmax.*(1 .- temp)
        end
        if boot
            return ParticleCollection([p0[:,k] for k in 1:Nparticles])
        else
            return WeightedParticleBelief([p0[:,k] for k in 1:Nparticles],ones(Nparticles)./Nparticles,1.0)
        end
    end

    function plotStep!(b,u,x,k,plots)
        plt = scatter(u[1,:], u[2,:], color=:red, label="UAS", xlim=(Xmin[1],Xmax[1]), ylim=(Xmin[2],Xmax[2]), ...
            legend=:bottomleft)
        for j in 1:maxUAS
            plot!(plt, u[1,j] .+ rMax*cos.(range(0,2*  ,500)),u[2,j] .+ ...
                rMax*sin.(range(0,2*  ,500)),linecolor=:green,label=nothing)
        end
        scatter!(plt,[p[1] for p in particles(b)], [p[2] for p in particles(b)], color=:black, markersize=1, label="")
        scatter!(plt,[x[1]], [x[2]], color=:blue, label="GT")
        scatter!(plt,[ParticleFilters.mode(b)[1]],[ParticleFilters.mode(b)[2]], color=:purple, markersize=2, ...
            label="Mode")
        # scatter!(plt,[ParticleFilters.mean(b)[1]],[ParticleFilters.mean(b)[2]], color=:pink, markersize=2, ...
            label="Mean")
        title!("Timestep: "*string(k))
        push!(plots, plt)
    end
#

if boot
    # PF = BootstrapFilter(ParticleFilterModel{Vector{Float64}}(dynPrediction,obsWeight), Nparticles)
    PF = BasicParticleFilter(PredictModel{Vector{Float64}}(dynPrediction), ReweightModel(obsWeight), ...
        LowVarianceResampler(Nparticles), Nparticles)
else
    PF = BasicParticleFilter(PredictModel{Vector{Float64}}(dynPredictionAux), ReweightModel(obsWeightAux), ...
        LowVarianceResampler(Nparticles), Nparticles)
end

# Simulation loop
    b = initBelief(Xmin,Xmax,Nparticles,boot) #initialize particle collection (belief) from uniform distribution
    measNoise = MvNormal(R) #AWGN distribution to sample from for measurement noise
    push!(xEst,rand(b)) # draw initial estimate from initial belief
    xEst2 = copy(xEst)
    for k in 1:endK
        # plot workspace and prior particle distribution
            p = plot(dpi=500)
            for j in eachindex(e.obs)
                plot!(p,e.obs[j])
            end
            scatter!(p,[p[1] for p in particles(b)], [p[2] for p in particles(b)], color=:black, markersize=1, label="")
            scatter!(p,[ParticleFilters.mode(b)[1]],[ParticleFilters.mode(b)[2]], color=:purple, markersize=2, ...
                label="Mode")
            if boot
                # scatter!(p,[ParticleFilters.mean(b)[1]],[ParticleFilters.mean(b)[2]], color=:pink, markersize=2, ...
                    label="Mean")
            end
        #

        # Get action (network configuration)
            if k == 1
                (xk,yk,xck,ck) = ...
```

```
                findStep(e,[gs[1];xEst[1][1];zeros(maxUAS,1)],[gs[2];xEst[1][2];zeros(maxUAS,1)],zeros(Ntot,1),1)
            else
                (xk,yk,xck,ck) = findStep(e,[gs[1];xEst[k][1] + ...
                    xEst[k][3];xPath[k-1][(Nga+1):Ntot]],[gs[2];xEst[k][2] + ...
                    xEst[k][4];yPath[k-1][(Nga+1):Ntot]],xcPath[k-1],k)
            end
        #

        # take measurement
            y = colwise(Euclidean(),xTrue[k][1:2],[xk[(Nga+1):Ntot]';yk[(Nga+1):Ntot]']) + 1 .*rand(measNoise)
            # send -1 measurement if out of range or LOS
            for j in 1:maxUAS
                if !inLOS(xTrue[k][1],xTrue[k][2],xk[j+Nga],yk[j+Nga],e.obs) || y[j] > rCon
                    y[j] = -1.0
                end
            end
            @show y
        #

        # update belief
        if boot
            global b = updateBoot(PF,b,[xk[(Nga+1):Ntot]';yk[(Nga+1):Ntot]'],y)
        else
            global b = updateAux(PF,b,[xk[(Nga+1):Ntot]';yk[(Nga+1):Ntot]'],y)
        end

        #Move forward in time and store timestep k values
            push!(xTrue,A*xTrue[k])
            push!(xPath,xk)
            push!(yPath,yk)
            push!(xcPath,xck)
            push!(cPath,ck)
            push!(xEst,ParticleFilters.mode(b))
            if boot
                # push!(xEst2,ParticleFilters.mean(b))
            end
        #

        # additional plotting
            scatter!(p,[xTrue[k][1]], [xTrue[k][2]], color=:red, label="GT", legend=:bottomleft)
            scatter!(p,[xPath[k][1]], [yPath[k][1]], color=:green, label="GS")
            scatter!(p,[xPath[k][2]], [yPath[k][2]], color=:blue, label="GT Prior")
            for j in 3:(maxUAS+2)
                scatter!(p, [xPath[k][j]], [yPath[k][j]], text = string(j-2), color=:yellow, label=nothing)
            end
            for j in 1:2
                plot!(p, xPath[k][j] .+ rCon*cos.(range(0,2*  ,500)),yPath[k][j] .+ ...
                    rCon*sin.(range(0,2*  ,500)),linecolor=:green,label=nothing)
            end
            for j in 1:Ntot
                for i in 1:Ntot
                    if(cPath[k][j,i] != 0) && xcPath[k][j] != 0
                        plot!(p,[xPath[k][j],xPath[k][i]], [yPath[k][j],yPath[k][i]],linecolor=:gray,label=nothing)
                    end
                end
            end
            xlims!(p,e.xlim[1]-1,e.xlim[2]+1)
            ylims!(p,e.ylim[1]-1,e.ylim[2]+1)
            title!(p,"Timestep: "*string(k))
            display(p);
            push!(plots,p)
            # savefig(p,"finalProject/plots/" * string(k) * ".png")
        #
        push!(err,sum((xEst[k][1:2] - xTrue[k][1:2]).^2))
    end
    # frames = Frames(MIME("image/png"), fps=1)
    # for plt in plots
    #     print(".")
    #     push!(frames, plt)
    # end
    # write("finalProject/plots/output.gif", frames)
#
```

`Workspace.jl`: used to generate the environment and obstacles, along with functions to interface with the environment.

```
module workspace #inspired by ASEN 5254 AMP-Tools-Public package by Peter Amorese
    # Collin Hudson 11/16/2024
    using StaticArrays
    using Random
    using LazySets
    using JuMP, HiGHS
```

```
export centroid
export shrink
export env
export inLOS
export inObs

function inLOS(p1x,p1y,p2x,p2y,obs)
    noHit = true
    signal = LineSegment([p1x,p1y],[p2x,p2y])
        for ob in obs
            vs = ob.vertices
            for j in 1:(length(vs)-1)
                if(noHit)
                    edge = LineSegment(vs[j],vs[j+1])
                    noHit = isdisjoint(signal, edge)
                else
                    return false
                end
            end
            if(noHit)
                edge = LineSegment(vs[end],vs[1])
                noHit = isdisjoint(signal, edge)
            else
                return false
            end
        end
    return true
end

function inObs(p1x,p1y,obs)
    for ob in obs
        if    ([p1x,p1y],ob)
            return true
        end
    end
    return false
end

function centroid(v::VPolygon)
    vts = [v.vertices;[v.vertices[1]]]
    centroid = [0.0;0.0]
    for j in 1:(length(vts)-1)
        centroid[1] += (vts[j][1] + vts[j+1][1])*(vts[j][1]*vts[j+1][2] - vts[j+1][1]*vts[j][2])
        centroid[2] += (vts[j][2] + vts[j+1][2])*(vts[j][1]*vts[j+1][2] - vts[j+1][1]*vts[j][2])
    end
    centroid = centroid/(6*area(v))
end

function shrink!(v::VPolygon,  ::Float64)
    c = centroid(v)
    for j in 1:length(v.vertices)
        v.vertices[j]  = v.vertices[j] +   *(c - v.vertices[j])
    end
    return v
end

function shrink(v::VPolygon,  ::Float64)
    v2 = copy(v)
    c = centroid(v2)
    for j in 1:length(v2.vertices)
        v2.vertices[j]  = v2.vertices[j] +   *(c - v2.vertices[j])
    end
    return v2
end

function checkArea(tempVec,xlim,ylim)
    tmpHull = convex_hull(tempVec[1],tempVec[2])
    for j = 3:length(tempVec)
        tmpHull = convex_hull(tmpHull,tempVec[j])
    end
    areaPoly = area(tmpHull)
    areaBounds = (xlim[2] - xlim[1])*(ylim[2] - ylim[1])
    if areaPoly > areaBounds
        return -1
    elseif areaPoly < 0.5*areaBounds
        return 1
    else
        return 0
    end
end

function scalePoly(v::VPolygon,xlim,ylim,)
```

```
        if diameter(v) > 0.75*min(xlim[2] - xlim[1],ylim[2] - ylim[1])
            return shrink!(v,0.5)
        else
            return v
        end
    end

    function polyMove(v::VPolygon,outVal)
            ang = -pi + 2*pi*rand()
            dirVec = [cos(ang);sin(ang)]
            return LazySets.translate(v,outVal*rand()*dirVec)
    end

    struct env
        xlim::SVector{2, Float64} #[min, max] limits for x coords
        ylim::SVector{2, Float64} #[min, max] limits for y coords
        obs #static vector of obstacles of type VPolygon (setting type caused bugs for some reason)
        Mobs #Vector of optimal M values
        Nverts #number of obstacle vertices
        function env(xlim,ylim,Nobs=5,seed=nothing)
            @assert xlim[2] > xlim[1] "Must have valid x bounds!"
            @assert ylim[2] > ylim[1] "Must have valid y bounds!"
            @assert Nobs >= 0 "Must have a non-negative number of obstacles!"
            if isnothing(seed)
                Random.seed!()
            else
                Random.seed!(seed)
            end
            #generate N obstacles with a random number of vertices (max 10)
            tempVec = []
            if Nobs > 0
                for j in 1:Nobs
                    push!(tempVec,LazySets.rand(VPolygon,num_vertices=rand(3:6)))
                end
                #Move all obstacles to be centered at origin of environment
                ctr = [sum(xlim)/2;sum(ylim)/2]
                tempVec = map(x->LazySets.translate(x,ctr-centroid(x)),tempVec)
                # Adjust size of polygons to fit bounds
                tempVec = map(x->scalePoly(x,xlim,ylim),tempVec)
                # Move polygons away from origin until footprint is large (or small) enough
                tooBig = checkArea(tempVec,xlim,ylim)
                while(tooBig != 0)
                    # Move polygons from environment origin a random distance in a random direction
                    tempVec = map(x->polyMove(x,tooBig),tempVec)
                    tooBig = checkArea(tempVec,xlim,ylim)
                end
                obsVec = SVector{Nobs,VPolygon}(tempVec)
                # Solve for optimal M values for obstacle avoidance
                Nverts = sum(x->length(x.vertices),obsVec)
                ks = zeros(Nverts,1)
                hs = zeros(Nverts,2*Nverts)
                row = 1
                for ob in obsVec
                    for h in LazySets.constraints_list(ob)
                        hs[row,:] = hcat(zeros(1,2*(row-1)),h.a',zeros(1,2*(Nverts - row)))
                        ks[row] = h.b
                        row += 1
                    end
                end
                model = Model(HiGHS.Optimizer)
                set_silent(model)
                @variable(model, x[i=1:2*Nverts])
                for j in 1:2:2*Nverts
                    @constraint(model,xlim[1] <= x[j] <= xlim[2])
                    @constraint(model,ylim[1] <= x[j+1] <= ylim[2])
                end
                @objective(model, Max, sum(ks - hs*x))
                optimize!(model)
                # Mobs = maximum(ks - hs*value.(x))*ones(length(value.(x)),1)
                Mobs = ks - hs*value.(x)
            else
                obsVec = SVector{Nobs,VPolygon}([])
                Mobs = []
                Nverts = 0
            end
            new(xlim,ylim,obsVec,Mobs,Nverts)
        end
    end

end
```