UavSim: An Open-Source Simulator for Multiple UAV Path Planning

Kyle Thompson
Computer Science
California Polytechnic State University
San Luis Obispo, USA
rkthomps@calpoly.edu

Dominik Walter
Computer Science
California Polytechnic State University
San Luis Obispo, USA
dowalter@calpoly.edu

Roman Maksymiuk
Computer Science
California Polytechnic State University
San Luis Obispo, USA
rmaksymi@calpoly.edu

Roey Mevorach
Computer Science
California Polytechnic State University
San Luis Obispo, USA
rmevorac@calpoly.edu

Gaurav Joshi
Computer Science
California Polytechnic State University
San Luis Obispo, USA
gjoshi@calpoly.edu

***Abstract*—Though the primary method for evaluating multiple UAV path planning algorithms is simulation, there is no open-source software built to compare algorithms. As a result, most researchers develop their own simulation environments. The presence of many simulation environments makes evaluation of separately developed algorithms difficult. To introduce standardization into the multiple UAV path planning space, we have created an open-source simulator for both development and evaluation of path planning algorithms. Our simulator focuses on the problem of small object detection using multiple UAVs. Its careful object-oriented design allows users unlimited flexibility in developing planning algorithms. UavSim is freely available on GitHub (https://github.com/rmaksymiuk/UavSim).**

***Index Terms*—Unmanned aerial vehicle, path planning, small-object detection, simulation, open-source**

I. Introduction

Increased quality of consumer grade drones, and continuous improvement in computer vision have made it feasible to use affordable UAVs to identify objects of interest through aerial surveillance [1]. Object detection using UAVs lies at the intersection of Computer Vision, and complete coverage path planning. Both topics have been well-studied individually [1], [2]. However, much less work has been done on the problem of efficiently covering an area of interest while achieving good performance in object detection. The most recent developments in this field propose strategies not covered in coverage path planning literature that sacrifice energy costs for object detection performance [3], [4].

At Cal Poly, there is an effort to use UAVs to spot sharks in the waters of the central coast. The first task for the shark-spotting project to achieve good performance using a single UAV, but there are plans to extend the scope of the project and use multiple UAVs to observe a known area. Our original goal was to investigate state of the art methods for small object detection with UAVs. We wanted to compare these methods through simulation. However, we were not able to find a simulation tool built for the problem of small object detection. We then had the idea of repurposing a simulation environment built solely for path planning. We were very surprised when we could not find a such an environment. After digging deeper into the literature on path planning for coverage, it was evident that there was no commonly used simulation software. We recognized an opportunity in both the small object detection and path planning spaces for a centralized simulation environment. Such an environment would give the topics' researchers an excellent basis for investigating strategies, and would especially help surveys like [2].

II. Related Work

To find related work, we sought to identify coverage path planning research that used simulation software that was not self-developed. We figured that we could look for this kind of research in a recent survey of path planning strategies. Cabreira, Brisolara, and Paulo [2] list 17 methods for path planning that either use no composition of the environment, or exact cellular decomposition of the environment. Of the 17 methods listed, there were only two instances where researchers used software that they did not develop themselves.

One research group at MgGill University published two papers where they used a UAV simulation software called Aviones [5], [6]. Both papers describe a Boustrophedon Cellular Decomposition (BCD) algorithm for non-holonomic vehicles that minimizes the distance travelled over previously covered regions. The researchers use Aviones to measure flight time and distance for UAVs in environments with and without wind [5], [6]. Aviones is a UAV Flight Simulator developed by the Brigham Young University Human Centered Machine Intelligence and Multiple Agent Intelligent Coordination and Control labs [7]. The software can accurately simulate autonomous navigation of an environment by UAVs where the environment can include wind [5]. However, the documentation for Aviones is severely limited, and their website has not been updated since 2007 [7].

Additionally, a research group at the University of Pennsylvania developed a leader-follower strategy for the localization of UAVs [8]. To test their strategy, the group employed a simulator developed in MATLAB [8]. The user interacts with the simulator through an event script [8]. This script is a matrix where each row represents a period of time [8]. Inside each row is the thrust, pitch and role the UAV should apply during that period of time [8]. A group at The University of Seville leveraged this simulation tool to simulate a decentralized multiple UAV coverage strategy [9]. The tool is able to handle a setting with obstacles and multiple UAVs [9].

The software presented in [8] is very capable. It simulates the motion of each UAV through its thrust and can model obstacles. However, it has a similar problem to Aviones. There is no central repository where a researcher could download or learn about it. We predict that the group at the University of Seville had direct communication with the group at the University of Pennsylvania to obtain the software.

Our work is significantly different than the Aviones platform, or the software used in [8]. We developed our tool for broad use. Our simulator is hosted on a public GitHub repository. Also on UavSim's repository is documentation on how to install and use the software. Our work focuses on the problem of small object detection, but could be easily used for coverage-focused applications. Neither Aviones, or the software used in [8] have capabilities suitable for small object detection.

III. Simulation Implementation

UavSim models the problem as an interaction between four primary entities. The UAV entity is a representation of a single UAV, or drone in the simulation. The object entity is a representation of a single object in the simulation. This object may or may not be the object of interest for a particular UAV. The environment entity conducts the simulation. It has knowledge about the setting of the simulation, and all other entities. The environment entity uses the plan entity to generate paths for the UAV entities. UavSim represents a path with a stack of waypoints, $[\vec{p_1}, \vec{p_2}, ..., \vec{p_n}]$, and a stack of speeds, $[s_1, s_2, ..., s_n]$. Both stacks always have equal length. At any point in time, if a UAV has a non-empty path, it pursues a position of $\vec{p_0}$ and a speed of s_0. We describe each entity in more detail in the remaining portion of this section.

A. Environment Entity

The environment entity first directs the plan entity to produce an initial path for each UAV. It then increments through time by a particular time step, Δt_{step}. At each time step, the environment entity updates its own state, and directs the UAV and object entities to update their respective states. If any of the UAVs in the environment detect an object while they are updating their state, the environment entity probes the plan entity to update the paths for each UAV. The simulation proceeds until each UAV has an empty path. In Algorithm 1, the environment entity is assumed to have the following attributes.

- *Uavs*: List of UAV entities in the environment.
- *Objects*: List of object entities in the environment.
- *Plan*: Plan entity.

Algorithm 1 Algorithm for Environment Entity

```
Paths = Plan.GetInitialPaths()
UpdateUAVPaths(Paths)
while Any UAV has a non-empty path do
  for Object in Objects do
    Object.Step(Δt_step)
  end for
  AllObjectsSpotted = []
  for Uav in Uavs do
    ObjectsSpotted = Uav.Step(Δt_step)
    if NonEmpty(ObjectsSpotted) then
      AllObjectsSpotted.append(uav, ObjectsSpotted)
    end if
  end for
  if NonEmpty(AllObjectsSpotted) then
    Paths = Plan.GetUpdatedPaths(AllObjectsSpotted)
    UpdateUAVPaths(Paths)
  end if
end while
```

B. UAV Entity

The UAV entity is the most complicated. It first needs to update its thrust to ensure that it is both pursuing a position of $\vec{p_0}$ and a speed of s_0. In other words, if the UAV entity has position $\vec{pos}$, the UAV entity needs to adjust its thrust to accelerate towards $\vec{vel_{desired}}$ where $\vec{vel_{desired}}$ is defined in (1).

$$\vec{vel}_{\text{desired}} = s_0\left(\frac{\vec{p_0 - pos}}{||p_0 - pos||}\right) \tag{1}$$

To simplify the calculation for finding a thrust that moves the UAV entity's velocity closer to $\vec{vel}_{\text{desired}}$, we define an algorithm that pursues $\vec{vel_{desired}}$ with the maximum possible acceleration. This algorithm is not guaranteed to minimize the thrust used between waypoints, but it is guaranteed to reach $\vec{vel}_{\text{desired}}$ in the minimum time possible. Forsmo et al. [10] provides a more computationally expensive way to find the optimal trajectory between waypoints. In Algorithm 2, the UAV entity is assumed to have the following attributes.

- $\vec{vel}$: Current velocity of the UAV entity.
- $\vec{F}_{\text{env}}$: Force vector of the environment normally just including gravity.
- $\vec{F}_{\text{tot}}$: Combined force vector of the environment and drag. If c_{area} is the estimated cross-sectional area for the drone, then $\vec{F}_{\text{tot}}$ is defined in (2). We use an estimate of 1.225 for ρ and and estimate of 1.2 for c_{d}.

$$\vec{F}_{\text{tot}} = \vec{F}_{\text{env}} + \frac{1}{2}\rho(-1 * \vec{vel})^2 c_{\text{d}} c_{\text{area}}. \tag{2}$$

- $\vec{t}_{\text{MaxForward}}$: Maximum forward thrust. Forward is the direction of $\vec{proj}_{\text{surface}}(\vec{vel})$ as defined in (3).

$$proj_{\text{surface}}(\vec{vel}) = \vec{vel} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{3}$$

- $\vec{t_{\text{MaxLeft}}}$: Maximum left thrust. Left is the direction of $proj_{\text{surface}}(\vec{vel})$ rotated °90 counter clockwise about the z axis.
- $\vec{t_{\text{MaxBackward}}}$: Maximum backward thrust. Backward is the direction of $proj_{\text{surface}}(\vec{vel})$ rotated °180 counter clockwise about the z axis.
- $\vec{t_{\text{MaxRight}}}$: Maximum right thrust. Right is the direction of $proj_{\text{surface}}(\vec{vel})$ rotated °270 counter clockwise about the z axis.
- $\vec{t_{\text{MaxUp}}}$: Maximum upward thrust. Upward is the direction of the standard basis vector $\vec{z}$.
- $\vec{t_{\text{MaxDown}}}$: Maximum downward thrust. Downward is the direction of the standard basis vector $\vec{z}$ rotated °180 about the x axis.
- TVecs:$[\vec{t_{\text{MaxLeft}}}, \vec{t_{\text{MaxBackward}}}, ..., \vec{t_{\text{MaxDown}}}]$

Algorithm 2 Algorithm to update UAV Thrust

```
Thrust = 0⃗
VelDiff = vel⃗_desired − vel⃗
ToCorrect = F⃗_tot
EnvContrib = proj_VelDiff(F⃗_tot)
if EnvContrib · VelDiff > 0 then
    ToCorrect = ToCorrect - EnvContrib
end if
for Tv⃗ in TVecs do
    EnvProj = proj_ToCorrect(Tv⃗)
    if EnvProj · ToCorrect < 0 then
        TvContrib = min(||EnvProj||, ||ToCorrect||) EnvProj/||EnvProj||
        Thrust = Thrust + TvContrib
        Tv⃗ = Tv⃗ - TvContrib
        ToCorrect = ToCorrect + TvContrib
    end if
    VelProj = proj_VelDiff(Tv⃗)
    if VelProj · VelDiff > 0 then
        Thrust = Thrust + VelProj
    end if
end for
return Thrust
```

Algorithm 2 calculates the change in velocity needed between the current velocity and the desired velocity. Then it checks whether environmental forces can contribute to necessary change in velocity. If they can, then the algorithm does not need to apply thrust to cancel out the entire environmental force vector, only the component orthogonal to the necessary change in velocity. The algorithm then iterates through thrust vectors of the UAV entity. For each vector, it checks to see if the vector can cancel out the outstanding environmental force. If it can, the projection of the thrust vector onto the outstanding environmental force vector is added to the UAV entity's thrust until either environmental force vector is $\vec{0}$, or the projection of the thrust vector is $\vec{0}$. If what remains of the thrust vector has a component in the direction of the necessary change in velocity, then that component it is added to the UAV entity's thrust.

After the UAV entity calculates its thrust, its other operations at each time increment are relatively straight forward. At a high level, the UAV needs to update its kinematics and performance metrics with respect to the time increment. Then, it needs to check if it reached p_0, the waypoint at the top of its path stack. It also needs to check if it spotted an object during the time increment. Algorithm 3 outlines the operations for a UAV entity. It assumes that the entity has the following attributes.

- $\vec{Thrust}$: UAV entity's current thrust as calculated in Algorithm 2.
- Path: The current path entity for the UAV entity.
- TimeSinceFrame: Total time elapsed since the UAV entity last captured an image.
- FPS: Frames per second for the UAV entity.
- TotalEnergyExpended: Aggregator that keeps track of the amount of energy used by a UAV.
- TotalAreaCovered: Polygon representing the union of all surfaces captured by UAV images.
- TPs, FPs, FNs: Counts associated with true positives, false positives, and false negatives respectively.
- SpeedCost: Function that takes a thrust vector as input and returns a rate of energy consumption.
- ObjectDetected: Function that takes a UAV entity and an object entity as input and returns one the four tuples below.
  - ("TP", Object Position): A tuple indicating that the UAV spotted the object, and the object was the object of interest.
  - ("FP", Object Position): A tuple indicating that the UAV spotted the object, but the object was not the object of interest.
  - ("FN", $Null$): A tuple indicating that the UAV did not spot the object, but the object was in the frame of the UAV, and the object was the object of interest.
- θ_{lat} and θ_{hor}: Angles associated with the UAV entity's camera as shown in Fig. 1. The region of surface coverage can be determined by pos, θ_{lat} and θ_{hor}.
- Objects: List of all object entities in the environment.

The UpdateUntilDesired function used in Algorithm 3 accepts a current vector, a desired vector, a rate of change vector, and Δt_{step}. If applying the rate of change vector to the current vector over Δt_{step} passes the desired vector, then the function returns the desired vector. If the desired vector was not passed, then the function returns the result of applying the rate of change for Δt_{step}. Using this function ensures that a UAV entity does not overshoot its target position or velocity. However, it introduces some inaccuracy. Essentially, the UAV ignores the rate of change vector in favor of a rate of change of $\vec{0}$ for the portion of Δt_{step} after the UAV reaches its desired vector. This inaccuracy decreases as Δt_{step} decreases, but as

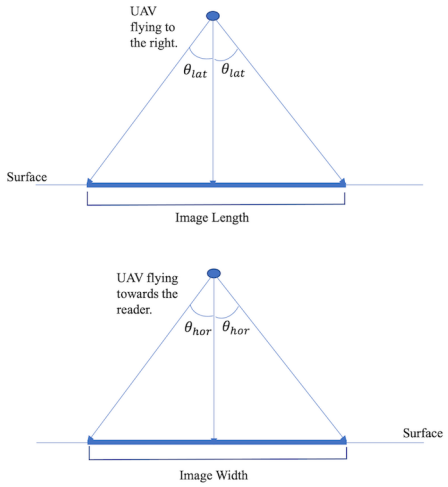

Fig. 1. Diagram of θ_{lat} and θ_{hor}

Δt_{step} decreases, computational expenses increase.

C. Object Entity

The object entity is much simpler than the UAV entity or the environment entity. Currently, we assume that an object has a constant velocity. Therefore, the object entity only needs to worry about maintaining its position throughout the simulation. Algorithm 4 shows the algorithm for an object entity assuming that it has the attributes $\vec{vel}$ and $\vec{pos}$ which mirror the $\vec{vel}$ and $\vec{pos}$ attributes from the UAV entity.

D. Plan Entity

Unlike the environment, UAV, or object entities, the plan entity is not defined by its behavior at each time increment in the simulation. Rather, the plan entity can be thought of as an interface between the user and the simulator. Like much of the research in [2], UavSim implements path planning algorithms by defining waypoints for UAVs. The purpose of the plan entity is to define the initial waypoints for each UAV entity, and to optionally redefine waypoints for each UAV entity whenever any UAV entity spots an object.

IV. Usage and Primary Features

In this section, we describe the usability of UavSim. UavSim allows the user to implement multiple UAV path planning algorithms by implementing the GetInitialPaths and GetUpdatedPaths functions for a plan entity. After the user defines his or her plan, the user must configure his or her environment. There are three steps to configuring the environment. First, the user must provide the specifications of each UAV Entity in the simulation. Specifically, the user must provide the following attributes.

- A unique name for the UAV entity.
- The type of object that the UAV entity is attempting to detect.
- The frames per second for the UAV entity.

Algorithm 3 Algorithm for UAV entity

```
Thrust = UpdateThrust()
v⃗el, ReachedVelGoal = UpdateUntilDesired(v⃗el, v⃗el_desired, Thrust, Δt_step)
p⃗os, ReachedPosGoal = UpdateUntilDesired(p⃗os, p⃗_0, v⃗el, Δt_step)
if ReachedPosGoal then
    Pop(Path.points)
    Pop(Path.speeds)
end if
TotalEnergyExpended = TotalEnergyExpended + Δt_step * SpeedCost(Thrust)
TimeSinceFrame = TimeSinceFrame + Δt_step
if TimeSinceFrame > FPS then
    TimeSinceFrame = 0
    Frame = GetCurrentFrame(pos, θ_lat, θ_hor)
    TotalAreaCovered = Union(TotalAreaCovered, Frame)
    ObjectsSpotted = []
    for Object in Objects do
        DetectType, DetectPos = ObjectDetected(Object)
        if DetectType = "TP" then
            TPs = TPs + 1
            ObjectsSpotted.Append(DetectPos)
        else if DetectType = "FP" then
            FPs = FPs + 1
            ObjectsSpotted.Append(DetectPos)
        else if DetectType = "FN" then
            Fns = FNs + 1
        end if
    end for
end if
return ObjectsSpotted
```

Algorithm 4 Algorithm for Object Entity

$\vec{pos} = \vec{pos} + \Delta t_{step} * \vec{vel}$

- The lateral and horizontal camera angles, θ_{lat} and θ_{hor}, for the UAV entity .
- The estimated cross-sectional area, c_{area}, of the UAV entity.
- The maximum up, down, left, right, forward, backward thrusts as scalars for the UAV entity.

The user can optionally provide the SpeedCost, and ObjectDetection functions described in the UAV Entity section. The user must also create object entities for the simulation. Each object entity requires a name, initial position, type, and velocity. UavSim has a utility function to automatically generate a given number of objects of a given type. Finally, the user must specify the environment. The user must provide the UAV entities, and the object entities that he or she created in addition to the boundary of the environment, the location of the control station, Δt_{step}, and an instantiation of the user defined plan.

After the user specifies and runs the simulation, UavSim

provides a number of statistics summarizing object detection performance and energy usage in the simulation. Table I shows an example of the statistics outputted by a sample simulation with three UAVs. The precision, recall, and f1 score are calculated with respect to the detection performance on the object of interest for each UAV respectively. The total precision, recall, and f1 score are found by summing true positives, false positives, and false negatives over all UAVs in the simulation. Covered is the calculated according to (4) if cov is the abbreviation of TotalAreaCovered, and $bound$ is the polygon associated with the environmental boundary. Spotted is the proportion of the objects of interest that were spotted by the UAV.

$$\text{Covered} = \text{Area}(cov \cap bound)/\text{Area}(bound) \tag{4}$$

TABLE I
SAMPLE SIMULATION OUTPUT

Name	Precision	Recall	F1	Energy	Covered	Spotted
uav1	0.951	0.868	0.908	893	0.351	0.400
uav2	0.934	0.768	0.843	1080	0.552	0.600
uav3	0.810	0.671	0.734	1220	0.552	0.600
Total	**0.898**	**0.768**	**0.828**	**3190**	**0.912**	**1.00**

UavSim also compiles a video that gives the user an excellent sense of the behavior of each UAV in the simulation. OpenCV compiles an .mp4 file from a series of MatPlotLib figures saved during the simulation [11], [12]. We've chosen to omit a still image of the video from this report because it is difficult to track the objects unless the viewer can see them moving. The reader can find a sample video on this project's Github repository.

V. EVALUATION

The obvious way to evaluate a simulation software is to compare its reported metrics to measurements taken in field tests. However, at this time, the Shark Spotting Project at Cal Poly is not ready to conduct autonomous flight tests. In lieu of a field test comparison, our team implemented three common coverage algorithms to ensure that our software produced sensible results. Each coverage algorithm is conducted with 3 UAVs that have identical specifications shown in Table II. Each simulation is run in an 1000m x 1000m environment with five sharks, three fish, and eight rocks. Fish and rocks are included as objects to allow opportunity for false positives. The Δt_{step} for each simulation is 0.01. Each simulation is run on a 2016 MacBook Pro with a 2.9Ghz Quad-Core Intel Core i7 processor. Plotting is turned off for these simulations to increase efficiency.

A. Boustrophedon Decomposition

BCD is a path planning strategy that divides an environment into cells and directs the UAVs in the environment to visit the cells in a snake-like fashion [13]. To implement BCD in UavSim, we choose a height for the UAVs to traverse the environment. At a given height, we can find the image size for each UAV entity. We use this image size to decompose a rectangle that is overlaid onto the environment's surface. The encompassing rectangle is broken into a grid of squares where the edge length of a square is defined by (5).

TABLE II
EVALUATION UAV CONFIGURATION

Attribute	Value
Focus	"Shark"
FPS	30
Mass	2kg
θ_{lat}	0.785rad
θ_{hor}	0.785rad
c_{area}	0.1m^2
Forward Thrust	40
Backward Thrust	5
Upward Thrust	40
Downward Thrust	0
Left Thrust	40
Right Thrust	40

$$\begin{aligned} Lengths &= \{Length(GetFrame(U))|U \in UAVs\} \\ Widths &= \{Width(GetFrame(U))|U \in UAVs\} \\ EdgeLength &= \min_{e}(e \in Lengths \cup Widths) \end{aligned} \tag{5}$$

We find the minimum set of squares that cover the environment's surface by dropping all squares from the grid that do not intersect with the environment's surface. If each remaining square can be represented by its row and column indices, $[i, j]$, then we can order the squares so that one square, s_1 precedes another square, s_2 if and only if Algorithm 5 returns True. We can then assign the squares to the UAVs such that if we have n UAVs, and m squares, we assign $\lceil \frac{m}{n} \rceil$ squares from the ordered list to each UAV until we run out of squares. We then create a path for each UAV by using the centroids of the squares as waypoints. For each waypoint we assign a constant speed of 20 m/s.

Algorithm 5 $s_1 < s_2$

```
ComesFirst = s_{1i} − s_{2i}
if ComesFirst = 0 then
    if IsEven(s_{1i}) then
        ComesFirst = s_{1j} − s_{2j}
    else
        ComesFirst = s_{2j} − s_{1j}
    end if
end if
return ComesFirst < 0
```

This is a very basic implementation of BCD. We do not decompose the region into convex polygons and calculate the major axis of each polygon. Instead, we direct the UAVs to always perform a snaking motion parallel to the y axis for the sake of simplicity. This basic implementation suffices for comparison because it is the basis for the following two

techniques. Table III contains the metrics UavSim recorded for BCD.

TABLE III
BCD RESULTS

Name	Precision	Recall	F1	Energy	Covered	Spotted
uav1	0.946	0.584	0.722	11001	0.325	0.600
uav2	0.943	0.637	0.760	12722	0.345	0.600
uav3	0.588	0.563	0.576	12906	0.348	0.200
Total	**0.891**	**0.600**	**0.717**	**36629**	**0.949**	**1.00**

B. Altitude Variant Boustrophedon Decomposition

Altitude Variant BCD (AVBCD) extends the cellular decomposition strategy. BCD is not oriented towards the problem of small object detection. It is solely focused on coverage efficiency. However, the performance of aerial object detection is related to the altitude of the UAV [3]. Therefore, if a UAV detects an object with some degree of certainty at a high altitude, it can decrease its altitude to obtain a higher degree of certainty [3].

To implement AVBCD, we simply added an implementation of the GetUpdatedPaths function to the BCD implementation. The previous implementation of the function simply returned an empty list which directs the UAVs in the simulation pursue their initial paths to exhaustion. The new implementation directs the UAV to drop to an altitude of 15m at a speed of 20m/s whenever it detects an object of interest. We can implement this behavior by letting UAV_i be an arbitrary UAV that spotted an object of interest at coordinates (x, y). All we have to do is push the point $(x, y, 15)$ onto the path for UAV_i with a corresponding speed of 20m/s. Table IV shows the metrics UavSim recorded for AVBCD.

TABLE IV
ALTITUDE VARIANT BCD RESULTS

Name	Precision	Recall	F1	Energy	Covered	Spotted
uav1	0.885	0.646	0.747	13085	0.325	0.600
uav2	0.603	0.660	0.630	14384	0.346	0.600
uav3	0.997	0.582	0.735	14085	0.348	0.600
Total	**0.838**	**0.620**	**0.713**	**41555**	**0.944**	**1.00**

C. Base Drone Boustrophedon Decomposition

Base Drone BCD (BDBCD) is a very loose interpretation of the work done in [4]. Like the altitude variant strategy, the role-based strategy is focused on optimizing both coverage, and object detection performance [4]. However, unlike the altitude variant strategy, the role-based strategy requires an estimated object-density distribution over the environment [4]. The density distribution is used to segment the environment into high-density areas and low-density areas [4]. "Explore" drones traverse low density areas at a high altitude to ensure efficient coverage of a potentially large area [4]. "Traverse" drones navigate high-density areas at a moderate altitude and direct "Observe" drones to visit each potential object of interest at a low altitude to ensure strong object detection performance in high density regions [4].

We implement a loosely related strategy that includes just two roles. The "Observer" drones are assigned initial paths in the exact same manner as BCD. However, the "Base" drones are not assigned paths upon instantiation of the simulation. Rather, whenever an "Observer" UAV detects an object of interest, it directs a "Base" drone that is not currently busy to visit the object and then return to the control station. If we let $(x_{\text{cont}}, y_{\text{cont}})$ be the coordinates of the control station, we create a path for a non-busy "Base Drone" consisting of the points $[(x, y, 15m), (x_{\text{cont}}, y_{\text{cont}}, 0)]$ at speeds [20m/s, 20m/s] respectively. Table V shows the metrics UavSim recorded for BDBCD where uav1 and uav2 are "Observer" drones, and uav3 is a "Base" drone.

TABLE V
BASE DRONE BCD RESULTS

Name	Precision	Recall	F1	Energy	Covered	Spotted
uav1	0.908	0.697	0.789	14545	0.475	0.400
uav2	0.844	0.493	0.623	15982	0.489	0.600
uav3	0.939	0.830	0.882	18640	0.081	0.600
Total	**0.904**	**0.673**	**0.772**	**49168**	**0.949**	**0.800**

D. Analysis

In comparing BCD, AVBCD, and BDBCD, we would expect that BCD uses less energy than both AVBCD, and BDBCD. When we mention energy, we really mean total impulse, which is measured in newton-seconds. Total impulse directly corresponds with the energy output of the drone. However, this correspondence is specific to the drone, which is why we provide a speed cost function for customization. In any case, BCD reports a total impulse of 36,629 N·s which is less than the 41,555 N·s reported by AVBCD and the 49,168 N·s reported by BDBCD. We would also expect BCD to report an inferior object detection performance than the other two strategies, but this is not necessarily the case. BCD reports an f1 score of 0.717 which is worse than BDBCD's f1 score of 0.772, but better than AVBCD's f1 score of 0.713. This counterintuitive result is a product of the logic in our ObjectDetected function. We compose a performance score used to determine whether or not an object was detected based on the height of the UAV, speed of the UAV, height of the object, and location of the object in the frame. We are currently unsure if this method is consistent with object detection results gathered in field tests. We hope that the Shark Spotting project at Cal Poly will give us an opportunity to tune the simulator's object detection logic. Users can always provide a custom ObjectDetected function that is suited to their model's performance.

We can also evaluate the computational performance of our model. The real time, simulation time, and steps per second

(SPS) are reported in Table VI. The most important measure in this table is SPS because it generalizes to different values of Δt_{step}. The most interesting observation is that BDBCD has a much lower SPS than BCD or AVBCD. There are only two reasons for this discrepancy. Either the GetUpdatedPaths function is more expensive for BDBCD, or the GetUpdatedPaths function is called more often for BDBCD, or both. We haven't looked into either of these causes.

TABLE VI
COMPUTATIONAL EVALUATION

Plan	Real Time	Simulated Time	Steps Per Second
BCD	701	358	51
AVBCD	783	388	49
BDBCD	1997	470	24

VI. CONCLUSIONS AND FUTURE WORK

UavSim offers a previously unavailable open-source simulation environment to researchers in both the small object detection and coverage path planning spaces. An accessible simulation software lowers the barrier to entry for doing research in either space which enables faster progress in both. Wide adoption of UavSim would simplify comparison of separately-conducted research and help clarify the state of the art.

While we have succeeded in our initial goal, there remain features that would improve UavSim. First, we hope that the Shark Spotting Project at Cal Poly will give us an opportunity to tune the simulator's object detection logic. We currently recommend that the user provide his or her own ObjectDetected function that corresponds well with the performance of his or her model. However, for UavSim to be used to compare research across groups, this logic must be standardized. If the Shark Spotting Project does not give a sufficient opportunity to tune the ObjectDetected function, we could just as well adopt an implementation from a third party contributor to be the standard.

We also used a highly object oriented implementation in python. We worry that some of our design choices are slowing down the speed of our simulator. We could increase the speed of the simulator by reimplementing some aspects of the UAV entity using pure numpy computation at the cost of readability.

Additionally, we have yet to include obstacles or wind in our simulation. Wind is somewhat trivial to implement because it just requires modification of the drag calculation, but obstacles will require more consideration. Both are important for a complete UAV simulation software.

Lastly, we would like to evaluate our software against field tests to measure its accuracy. The Shark Spotting Project at Cal Poly will likely give us an opportunity to perform this evaluation.

REFERENCES

[1] P. Mittal, R. Singh, and A. Sharma, "Deep learning-based object detection in low-altitude UAV datasets: A survey," Image and Vision Computing, vol. 104, p. 104046, Dec. 2020, doi: 10.1016/j.imavis.2020.104046.

[2] T. Cabreira, L. Brisolara, and P. R. Ferreira Jr., "Survey on Coverage Path Planning with Unmanned Aerial Vehicles," Drones, vol. 3, no. 1, p. 4, Jan. 2019, doi: 10.3390/drones3010004.

[3] M. Krusniak, K. Leppanen, Z. Tang, F. Gao, Y. Wang, and Y. Shang, "A Detection Confidence-Regulated Path Planning (DCRPP) Algorithm for Improved Small Object Counting in Aerial Images," in 2020 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, Jan. 2020, pp. 1–6. doi: 10.1109/ICCE46568.2020.9043152.

[4] M. Krusniak, A. James, A. Flores, and Y. Shang, "A Multiple UAV Path-Planning Approach to Small Object Counting with Aerial Images," in 2021 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, Jan. 2021, pp. 1–6. doi: 10.1109/ICCE50685.2021.9427712.

[5] A. Xu, C. Viriyasuthee, and I. Rekleitis, "Optimal complete terrain coverage using an Unmanned Aerial Vehicle," in 2011 IEEE International Conference on Robotics and Automation, Shanghai, China, May 2011, pp. 2513–2519. doi: 10.1109/ICRA.2011.5979707.

[6] A. Xu, C. Viriyasuthee, and I. Rekleitis, "Efficient complete coverage of a known arbitrary environment with applications to aerial operations," Auton Robot, vol. 36, no. 4, pp. 365–381, Apr. 2014, doi: 10.1007/s10514-013-9364-x.

[7] "Aviones: UAV Flight Simulator," http://aviones.sourceforge.net. Accessed: 03/10/22.

[8] W. H. E. Jr, P. Martin, and R. Mangharam, "Cooperative Flight Guidance of Autonomous Unmanned Aerial Vehicles," p. 11.

[9] J. J. Acevedo, B. C. Arrue, I. Maza, and A. Ollero, "Distributed Approach for Coverage and Patrolling Missions with a Team of Heterogeneous Aerial Robots under Communication Constraints," International Journal of Advanced Robotic Systems, vol. 10, no. 1, p. 28, Jan. 2013, doi: 10.5772/52765.

[10] E. J. Forsmo, E. I. Grotli, T. I. Fossen, and T. A. Johansen, "Optimal search mission with Unmanned Aerial Vehicles using Mixed Integer Linear Programming," in 2013 International Conference on Unmanned Aircraft Systems (ICUAS), Atlanta, GA, May 2013, pp. 253–259. doi: 10.1109/ICUAS.2013.6564697.

[11] Gary Bradski, "The OpenCV Library," Dr. Dobb's Journal; San Mateo, vol. 25, no. 11, pp. 120–125, Nov. 2000.

[12] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," Comput. Sci. Eng., vol. 9, no. 3, pp. 90–95, 2007, doi: 10.1109/MCSE.2007.55.

[13] H. Choset, "Coverage of Known Spaces: The Boustrophedon Cellular Decomposition," p. 7.