Deep Learning Decision Making for Autonomous Drone Landing in 3D Urban Environment

Oren Gal[1,2] and Yerach Doytsher[3]

[1]Department of Marine Technologies, University of Haifa, Haifa, Israel
[2]Kinneret Academic College, Kinneret, Israel
[3]Mapping and Geo-information Engineering, Technion - Israel Institute of Technology, Haifa, Israel

e-mails: {orengal@alumni.technion.ac.il, doytsher@technion.ac.il}

***Abstract*— Quadcopters are four rotor Vertical Take-Off and Landing (VTOL) Unmanned Aerial Vehicle (UAV) with agile manoeuvring ability, small form factor and light weight – which makes it possible to carry on small platforms. Quadcopters are also used in urban environment for similar reasons – especially the ability to carry on small payloads, instead of using helicopters on larger vehicle which are not possible in these dense places. In this paper, we present a new approach for autonomous landing a quadcopter in 3D urban environment, where the first stage is based on free obstacle environment and maximal visibility for the drone in the palled landing spot. Our approach is based on computer-vision algorithms using markers identification as input for the decision by Stochastic Gradient Descent (SGD) classifier with Neural Network decision making module with greedy motion planner avoiding static and dynamic obstacles in the environment. We use OpenCV with its built-in ArUco module to analyse the camera images and recognize platform/markers, then we use Sci-Kit Learn implementation of SGD classifier to predict landing optimum angle and compare results to manually decide by simple calculations. Our research includes real-time experiments using Parrot Bebop2 quadcopter and the Parrot Sphinx Simulator.**

Keywords - Swarm; Visibility; 3D; Urban environment; autonomous landing.

I. Introduction and Related Work

A Quadcopter is a specific type of a UAV, with four rotors and Vertical takeoff and Landing (VTOL) capability, its agility, light weight and size makes it a perfect companion to smaller boats from sail-boats to even kayak, rather than classic helicopters that accompany bigger ships or fixed-wings airplanes on extremely large aircraft carriers.

The efficient computation of visible surfaces and volumes in 3D environments is not a trivial task. The visibility problem has been extensively studied over the last twenty years, due to the importance of visibility in GIS and Geomatics, computer graphics and computer vision, and robotics. Accurate visibility computation in 3D environments is a very complicated task demanding a high computational effort, which could hardly have been done in a very short time using traditional well-known visibility methods [1].

The exact visibility methods are highly complex, and cannot be used for fast applications due to their long computation time. Previous research in visibility computation has been devoted to open environments using DEM models, representing raster data in 2.5D (Polyhedral model), and do not address, or suggest solutions for, dense built-up areas.

Most of these works have focused on approximate visibility computation, enabling fast results using interpolations of visibility values between points, calculating point visibility with the Line of Sight (LOS) method. Lately, fast and accurate visibility analysis computation in 3D environments.

A vast number of algorithms have been suggested for speeding up the process and reducing computation time. Franklin evaluates and approximates visibility for each cell in a DEM model based on greedy algorithms. Wang et al. introduced a Grid-based DEM method using viewshed horizon, saving computation time based on relations between surfaces and the line of sight (LOS method). Later on, an extended method for viewshed computation was presented, using reference planes rather than sightlines.

One of the most efficient methods for DEM visibility computation is based on shadow-casting routine. The routine cast shadowed volumes in the DEM, like a light bubble. Extensive research treated Digital Terrain Models (DTM) in open terrains, mainly Triangulated Irregular Network (TIN) and Regular Square Grid (RSG) structures. Visibility analysis in terrain was classified into point, line and region visibility, and several algorithms were introduced, based on horizon computation describing visibility boundary.

In the many uses of UAV (Unmanned Aerial Vehicle) a pilot uses real-time telemetry to take-off, fly and land the craft with continuous communication between ground station and the UAV on-board computer. Making these tasks autonomous, will allow UAVs to perform missions without continuous communication, and thus prevent hijack or damage by hackers, be more stealth for surveillance and have unlimited distance from ground station (bound to energy limitation).

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Autonomous landing of a UAV is a problem on the focus of many studies [6][7][8] and landing on marine vessel makes this problem even more complex due to sea level motion that also occur when target platform is at stand-still.

The object of this research is to produce a safe landing mechanism for a quadcopter in 3D urban environment, in order to allow it to perform fully autonomous missions carried out at sea. Also, this mechanism could be used in pilot guided missions, as guideline suggestions to the pilot with how/when it is safe to land.

We assume the target position is known and Ground Station sets "home" position in the drone to be target's GPS position. Then the Bebop2 built-in "Return Home" function will bring it to the target, with up to a few meters off.

The proposed mechanism will perform the following tasks to achieve a "safe landing" decision: First, we need to visually search for and recognize the platform target and find the docking area. Once the target is found, the drone should set course and fly to target to be exactly above. Then, we detect and analyze the position of the landing surface and its plane angle relative to the camera. And finally, we will send the data to each of two implementations of the decision algorithms: 1. Using a supervised machine-learning classifier (pre-loaded with data), The machine input requires a quick pre-processing to set the data into a fixed structure vector, to resemble fitted data in the classifier. 2. Calculating directly from the data returned from the ArUco detection functions. The drone will then land safely on the boat, by sending a "land" command on time.

The problem of autonomous landing an UAV was on the focus of many studies as the survey review state-of-the art methods of vision-based autonomous landing, for a wide range of UAV classes from fixed-wing to multi-rotors and from large-scale aircrafts to miniatures. The main motivation for dealing with autonomous landing is the difficulty in performing a successful landing even with a pilot controlling the UAV. As it seems by statistics showed in [5], most of the accidents related to Remotely Piloted Aircraft Systems (RPAS) occur when the pilot tries to land the UAV.

Extensive research has been done on the subject to explore the various situations, technologies and methods to engage this problem. The work performed on previous studies, reviewed later in this section, is a great starting point for this project, as it is purely academic and relays on series of already existent technologies and tools, such as OpenCV [4], Sci-kit learn and the Parrot Ground SDK [2].

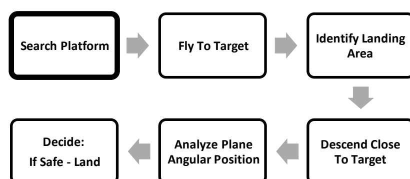


Figure 1. Proposed autonomous landing mechanism

In the following sections, we first introduce an overview of 3D models and extended the 3D visible volumes analysis. In the next section, we present the autonomous navigation process based on our fast visibility analysis with training data and classifier as can be seen in Figure 1. Later, we present the simulation based on our 3D visible volumes analysis.

II. AUTONOMOUS NAVIGATION PROCESS

The basic step starting this process related to obstacle avoidance and visible area described in the next sections. Following that, we divide the autonomous navigation mission into two separate problems. The first part deals with navigating UAV from an arbitrary position far from target, as far field. The second part is related to navigating to the target in the near field where the target is visible.

In the first scenario, which is when the mission objectives are reached and the drone needs to get to the target vessel for landing, we can use the built-in functionality of the drone to "Return Home" by setting it "Home" position to the target's known GPS position.

Bebop2 "Return Home" function works in a way that it will lift the drone to 20m above ground relative to take-off position, then fly directly to GPS position of "Home" and descend to 2m. Notice that if the drone is starting at height of more than 20m it will not descend to 20m, but rather keep its height until final descend near "Home".

The "Return Home" accuracy brings the drone to "Home" sometimes with offset of a few meters. This is good enough to get us to the second problem of navigation with visual distance to the target, until the drone will be directly above target and ready for landing.

Once the drone is at "Home" position, it will rotate and with each full rotation the tilt angle will increase to look further below, and if after rotating and tilting to the maximum of -90 degrees to the horizon, i.e., directly down, it will try again at higher altitude (1m up) to maybe see further away.

After getting a visual identification the drone will set course, keeping the target in the middle of the screen, and moving forward to it, tilting the camera during the movements until the landing pad is directly below. According to that, landing pad located in the middle of the image and camera tilt is maximum.

Then the drone will lower altitude to ~50cm while keeping the landing pad centered underneath, and in that height the data from the AR tags will be converted to a vector of predefined structure to feed a classifier trained to detect optimum landing angle/position. Once the classifier gives "Safe" signal – a "Land" command will issue to the drone to perform immediately.

III. FAST AND APPROXIMATED VISIBILITY ANALYSIS

In this section, we present an analytic analysis of the visibility boundaries of planes, cylinders and spheres for the predicted scene presented in the previous sub-section, which leads to an approximated visibility.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A. Analytic 3D Visible Volumes Analysis

In this section, we present fast 3D visible volumes analysis in urban environments, based on an analytic solution which plays a major role in our proposed method of estimating the number of clusters. We present an efficient solution for visible volumes analysis in 3D.

We analyze each building, computing visible surfaces and defining visible pyramids using analytic computation for visibility boundaries. For each object we define Visible Boundary Points (VBP) and Visible Pyramid (VP).

A simple case demonstrating analytic solution from a visibility point to a building can be seen in Figure 2(a). The visibility point is marked in black, the visible parts colored in red, and the invisible parts colored in blue where VBP marked with yellow circles.

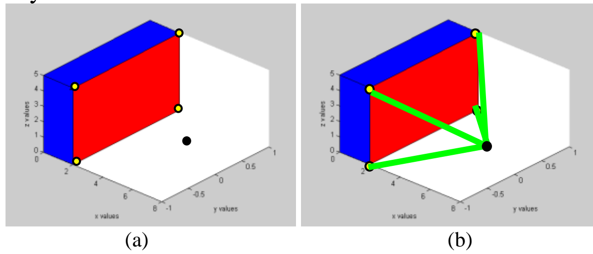


Figure 2. (a) Visibility Volume Computed with the Analytic Solution. (b) Visible Pyramid from a Viewpoint (marked as a Black Dot) to VBP of a Specific Surface

In this section, we introduce our concept for visible volumes inside bounding volume by decreasing visible pyramids and projected pyramids to the bounding volume boundary. First, we define the relevant pyramids and volumes.

The Visible Pyramid (VP): we define $VP_i^{j=1..N_{surf}}(x_0, y_0, z_0)$ of the object *i* as a 3D pyramid generated by connecting VBP of specific surface *j* to a viewpoint $V(x_0, y_0, z_0)$.

In the case of a box, the maximum number of N_{surf} for a single object is three. VP boundary, colored with green arrows, can be seen in Figure 2(b).

For each VP, we calculate Projected Visible Pyramid (PVP), projecting VBP to the boundaries of the bounding volume S.

Projected Visible Pyramid (PVP) - we define $PVP_i^{j..N_{surf}}(x_0, y_0, z_0)$ of the object *i* as 3D projected points to the bounding volume *S*, VBP of specific surface *j* trough viewpoint $V(x_0, y_0, z_0)$. VVP boundary, colored with purple arrows, can be seen in Figure 3.

The 3D Visible Volumes inside bounding volume *S*, VV_S, computed as the total bounding volume *S*, V_S, minus the Invisible Volumes IV_S. In a case of no overlap between buildings, IV_S is computed by decreasing the visible volume from the projected visible volume, $\sum_{i=1}^{N_{obj}} \sum_{j=1}^{N_{surf}} (V(PVP_i^j) - V(VP_i^j))$.

By decreasing the invisible volumes from the total bounding volume, only the visible volumes are computed, as seen in Figure 4. Volumes of VPV and VP can be simply computed based on a simple pyramid volume geometric formula.

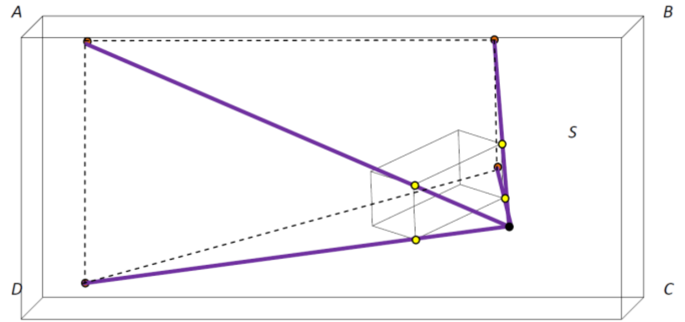


Figure 3. Invisible Projected Visible Pyramid Boundaries colored with purple arrows from a Viewpoint (marked as a Black Dot) to the boundary surface ABCD of Bounding Volume *S*

In a case of two buildings without overlapping, IV_S computed for each building, as presented above, as can be seen in Figure 5.

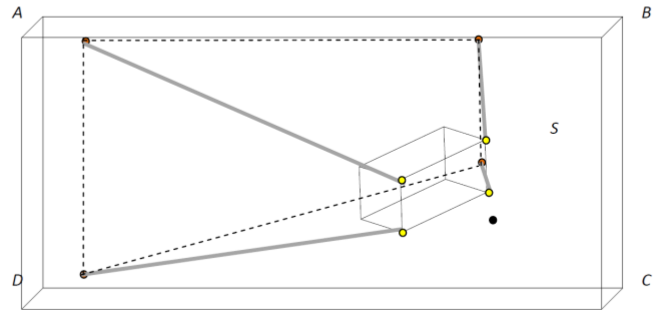


Figure 4. Invisible Volume $V(PVP_i^j) - V(VP_i^j)$ Colored in Gray Arrows. Decreasing Projected Visible Pyramid boundary surface ABCD of Bounding Volume S from Visible Pyramid

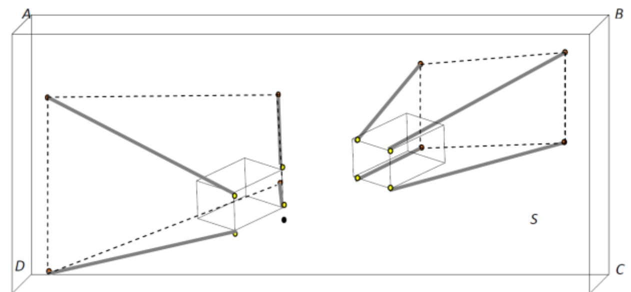


Figure 5. Invisible Volume $V(PVP_i^j) - V(VP_i^j)$ Colored in Gray Arrows. Decreasing Projected Visible Pyramid boundary surface ABCD of Bounding Volume S from Visible Pyramid

Considering two buildings with overlap between object's Visible Pyramids, as seen in Figure 6(a). In Figure 6(b), VP_1^1 boundary is colored by green lines, VP_2^1 boundary is colored by purple lines and the hidden and Invisible Surface between visible pyramids $IS_{VP_1^i}^{VP_2^i}$ is colored in white.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Invisible Hidden Volume (IHV) - We define Invisible Hidden Volume (*IHV*), as the *Invisible Surface (IS)* between visible pyramids projected to bounding box *S*.

For example, IHV in Figure 6(c) is the projection of the invisible surface between visible pyramids colored in white, projected to the boundary plane of bounding box *S*.

In the case of overlapping buildings, by computing invisible volumes IV_S, we decrease IHV twice between the overlapped objects, as can be seen in Figure 6(c), *IHV* boundary points denoted as $\{A_{11},..,A_{18}\}$. The same scene is presented in Figure 7, where Invisible Volume $V(PVP_i^j) - V(VP_i^j)$ is colored in purple and green arrows for each building.

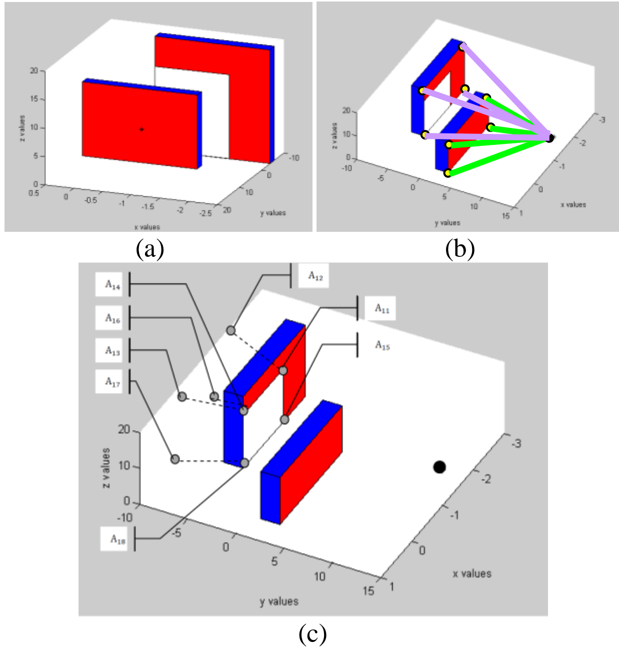


Figure 6. (a) Computing Hidden Surfaces between Buildings , VP_2^1 Base Plane, $IS_{VP_1^i}^{VP_2^i}$ (b) The Two Buildings - VP_1^1 in green and VP_2^1 in Purple (from the Viewpoint) and $IS_{VP_1^i}^{VP_2^i}$ in White (c) IHV boundary points colored with gray circles denoted

The *PVP* of the object close to the viewpoint is marked in black, colored with pink circles denoted as boundary set points $\{B_{11},..,B_{18}\}$ and the far object's *PVP* is colored with orange circles, denoted as boundary set points $\{C_{11},..,C_{18}\}$. It can be seen that *IHV* is included in each of these invisible volumes, where $\{A_{11},..,A_{18}\} \in \{B_{11},..,B_{18}\}$ and $\{A_{11},..,A_{18}\} \in \{C_{11},..,C_{18}\}$.

Therefore, we add *IHV* between each overlapping pair of objects to the total visible volume.

The same analysis holds true for multiple overlapping objects, adding the IHV between each two consecutive objects.

In Figure 8, we demonstrate the case of three buildings with overlapping. The invisible surfaces are bounded with dotted lines, while the projected visible surfaces to the overlapped building are colored in gray. In order to calculate the visible volumes from a viewpoint, *IHV* between each two buildings must be added as a visible volume, since it is already omitted at the previous step as an invisible volume.

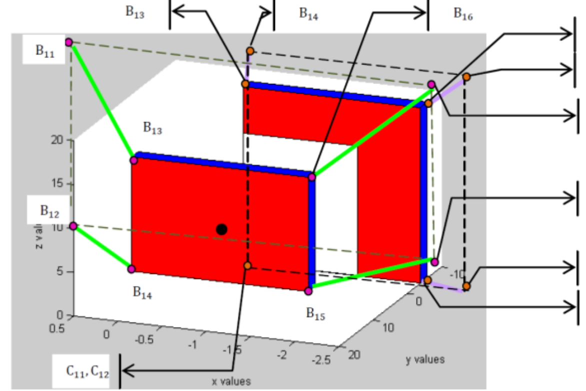


Figure 7. Invisible Volume $V(PVP_i^j) - V(VP_i^j)$ colored in purple and green arrows for each building. PVP of the object close to viewpoint colored in black, colored with pink circles and the far object PVP colored with orange circle

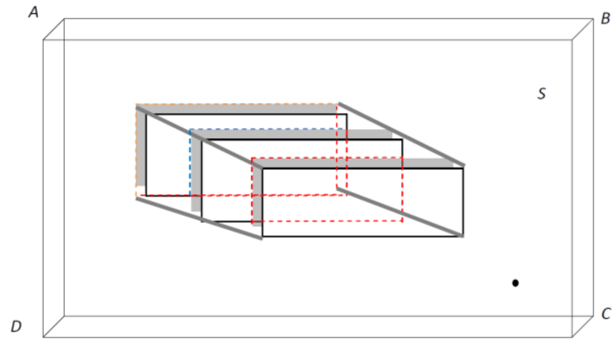


Figure 8. Three overlapping buildings. Invisible surfaces bounded with dotted lines, projected visible surfaces of the overlap building colored in gray

In this part, we extend the previous visibility analysis concept and include cylinders as continuous curves parameterization $C_{clnd}(x, y, z)$.

Cylinder parameterization can be described in (1):

$$C_{Clnd}(x,y,z) = \begin{pmatrix} r\sin(\theta) \\ r\cos(\theta) \\ c \end{pmatrix}_{r=const}, \quad \begin{matrix} 0 \le \theta \le 2\pi \\ c = c+1 \\ 0 \le c \le h_{peds_max} \end{matrix} \qquad (1)$$

We define the visibility problem in a 3D environment for more complex objects as:

$$C'(x,y)_{z_{const}} \times (C(x,y)_{z_{const}} - V(x_0, y_0, z_0)) = 0 \qquad (2)$$

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

where 3D model parameterization is $C(x,y)_{z=const}$, and the viewpoint is given as $V(x_0, y_0, z_0)$. Extending the 3D cubic parameterization, we also consider the case of the cylinder. Integrating (1) to (2) yields:

$$\begin{pmatrix} r\cos\theta \\ -r\sin\theta \\ 0 \end{pmatrix} \times \begin{pmatrix} r\sin\theta - V_x \\ r\cos\theta - V_y \\ c - V_z \end{pmatrix} = 0 \quad (3)$$

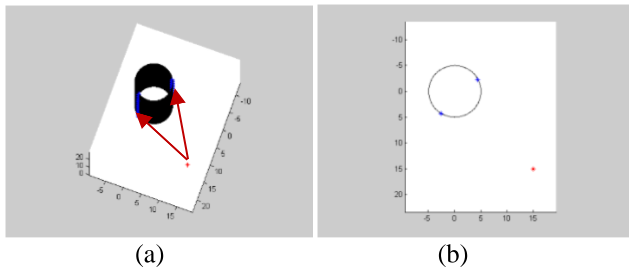
$$\theta = \arctan\left(-\frac{-r - \frac{\left(-vy\, r + \sqrt{vx^4 - vx^2 r^2 + vy^2 vx^2}\right) vy}{vx^2 + vy^2}}{vx}, -\frac{-vy\, r + \sqrt{vx^4 - vx^2 r^2 + vy^2 vx^2}}{vx^2 + vy^2} \right) \quad (4)$$

As can be noted, these equations are not related to Z axis, and the visibility boundary points are the same for each *x-y* cylinder profile, as seen in (3), (4).

The visibility statement leads to complex equation, which does not appear to be a simple computational task. This equation can be efficiently solved by finding where the equation changes its sign and crosses zero value; we used analytic solution to speed up computation time and to avoid numeric approximations. We generate two values of θ generating two silhouette points in a very short time computation. Based on an analytic solution to the cylinder case, a fast and exact analytic solution can be found for the visibility problem from a viewpoint.

We define the solution presented in (4) as x-y-z coordinates values for the cylinder case as Cylinder Boundary Points (CBP). CBP, defined in (5), are the set of visible silhouette points for a 3D cylinder, as presented in Figure 9:

$$CBP_{i=1..N_{PBP_bound}=2}(x_0, y_0, z_0) = \begin{bmatrix} x_1, y_1, z_1 \\ x_{N_{PBP_bound}}, y_{N_{PBP_bound}}, z_{N_{PBP_bound}} \end{bmatrix} \quad (5)$$



(a) (b)

Figure 9. Cylinder Boundary Points (CBP) using Analytic Solution marked as blue points, Viewpoint Marked in Red: (a) 3D View (Visible Boundaries Marked with Red Arrows); (b) Topside View.

In the same way, sphere parameterization can be described as formulated in (6):

$$C_{Sphere}(x,y,z) = \begin{pmatrix} r\sin\phi\cos\theta \\ r\sin\phi\sin\theta \\ r\cos\phi \end{pmatrix}_{r=const} \quad (6)$$
$$0 \le \phi < \pi$$
$$0 \le \theta < 2\pi$$

We define the visibility problem in a 3D environment for this object in (7):

$$C'(x,y,z) \times (C(x,y,z) - V(x_0, y_0, z_0)) = 0 \quad (7)$$

where the 3D model parameterization is $C(x,y,z)$, and the viewpoint is given as $V(x_0, y_0, z_0)$. Integrating (6) to (7) yields:

$$\theta = \arctan\left(\frac{r\sin(\phi)}{v_y} - \frac{1}{v_y\,(v_y^2 + v_x^2)}\left(v_x\left(r\sin(\phi)\, v_x - \sqrt{-v_y^2 r^2 \sin(\phi)^2 + v_y^4 + v_x^2 v_y^2}\right)\right), \frac{r\sin(\phi)\, v_x - \sqrt{-v_y^2 r^2 \sin(\phi)^2 + v_y^4 + v_x^2 v_y^2}}{v_y^2 + v_x^2} \right) \quad (8)$$

Where r is defined from sphere parameter, and $V(x_0, y_0, z_0)$ are changes from visibility point along Z axis, as described in (8). The visibility boundary points for a sphere, together with the analytic solutions for planes and cylinders, allow us to compute fast and efficient visibility in a predicted scene from local point cloud data, which are updated in the next state.

This extended visibility analysis concept, integrated with a well-known predicted filter and extraction method, can be implemented in real time applications with point clouds data.

IV. VISIBILITY-BASED DRONE AUTONOMOUS LANDING

The landing pad designed as a plate with five markers – one in the center and four others on each corner:

Figure 10. Landing pad with fiducial markers

Every ArUco marker has an ID as described in Figure 10, which can be determined when the marker gets detected, and

by that we can easily center the drone location above the landing pad even if only one or two markers are in view.

This landing pad has markers with ID values of {18,28, 17,25,4} selected randomly, but once selected they are very important to the implementation since the training data linked to the classifiers used as will be discussed later.

The proposed system takes each frame, and resolve all markers, then create a data vector of fixed length with all the necessary information of the markers.

Data format for each marker can be described as: [ID, rx, ry, rz, tx, ty, tz],

Where $\begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix}$ is the rotation vector of a single marker and $\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$ is the translation vector of that marker. This format repeats five times in each vector, where a tag ID has a fixed position for each tag. When a marker could not be found on a frame, the tag ID and all values of that marker will be set to zero.

Then send this vector to a classifier which will simply return strings telling us if the drone is centered above the landing pad or a correction movement is required. Possible answers are in the set: “CENTER”, “DOWNWARD”, “FORWARD”, “RIGHT”, “LEFT”.

For the Navigation we added more ArUCO tags surrounding this pad, in three sizes, so that they will be visible from varying distances along the navigation and descend process of the mechanism.

We used eight large tags, each surrounded by four medium tags and in between another five small tags as seen in Figure 11. The landing pad is printed on A4 page. And each of the eight patterns described here is also on an A4 page.

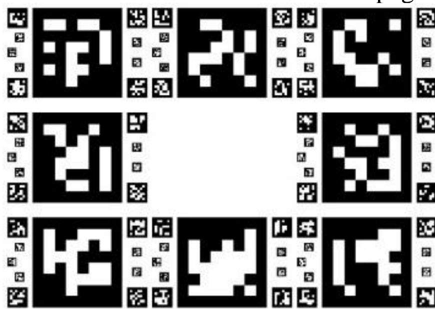


Figure 11. Navigation Assisting Tag Board Design

A. *Training Data and Classifiers*

In order to train the classifier, we used OpenGL as can be seen in Figure 12 to simulate the landing pad in a precisely controlled position and viewing angles. By that, we created a labeled data set, then use this precisely labeled data to fit in a variety of classifiers and test for accuracy. Following that, we tested several classifiers and selected the best performance for the purpose of the landing mechanism proposed.

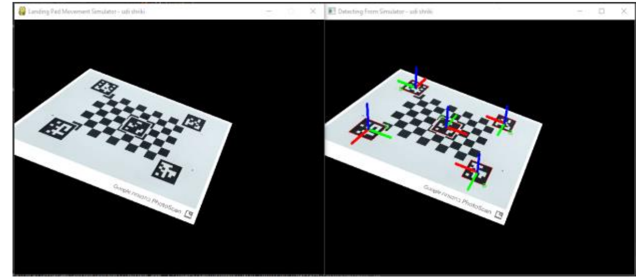


Figure 12. OpenGL Simulation for Training Data

The simulated platform rotating in roll, pitch and yaw - controlled by passing parameters, allowing me to tag every rendered frame as either safe for landing or not without visual computation (pre-label the data).

The simulator gets parameters from command line for setting some axis angle to run on a limited range, while rolling over all possible values of angles and positions, so the workload could be divided to parallel processes and even run on different machines.

After a few days running on several computers in parallel, the simulators generated a total of 15,193,091 vectors dataset, that could be used as training dataset for different models of classifiers.

Sci-Kit Learn package implements SVM with a fit function that takes labeled data as input in two variables: Y vector of y labels in a single column and X array of x vectors – each x vector is a line vector corresponds to the appropriate y in Y.

SVM does not allow incremental learning, i.e., it needs all data at once. This was quit an issue with the data size we tried to fit – fifteen million vectors. However, Sci-Kit Learn offers other types of classifiers, although all of them do not perform actual incremental learning (they do need all data at once), nonetheless, they do implement a partial fit function that can take each round a small portion of the data, and update the classifier’s support vectors.

For each classifier, we tried different parameters, and different sizes of the dataset by selecting randomly a fraction of the data. Then, test the model (using 25% of the data for test) to check it prediction accuracy.

TABLE I. CLASSIFIERS ACCURACY COMPARISSION

Classifier Type	Best accuracy
SGD, epsilon insensitive	57.341%
SGD, hinge	75.716%
SGD, huber	59.841%
SGD, log	73.658%
SGD, modified huber	73.362%
SGD, squared eps. insensitive	59.6%
SGD, squared hinge	73.857%
SGD, squared loss	57.171%
Perceptron	74.579%
Bernoulli NB	62.317%
Passive Aggressive Classifier	74.455%

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The result in Table I shows that even the best classifier got only approximately 75% success in recall. This is insufficient for a safety mechanism even with filters added to the process of a final “safe” decision.

To further increase accuracy, we thought it would be more effective to use more than one classifier, in a voting manner, to decide together on the data. At first, we suggested a voting scheme that takes 10-15 of the best classifiers and check if more than 50% of them agree on a "safe" result, take that as the answer, we checked that over the data and results did not increase accuracy at all. Then we thought maybe a classifier of classifiers outputs could extract some new information in a smarter manner than a simple voting, and will help increase accuracy. We created a new dataset of the same size, only this time the vector consisted of zero for safe and one for unsafe result of a classifier over fifteen of the best classifiers (72%-75% accuracy) and trained this dataset on all types of classifiers with different parameters as before. This time, all classifiers listed above got around 76% accuracy, where the best classifier reached 76.8% accuracy. Approximately 2% improvement.

Finally, looking closely on live videos of the ArUco markers detections, we noticed that the axis drawn on the detected markers tend to shift rapidly usually around more “safe” angles, so we tried to manually correct the data, and remove some of the spiking data that is tagged as safe – i.e., the simulator created it as a safe angle, but detection errors made it as a vector that should rather be tagged as unsafe.

All data marked as safe, with “Z” axis angle in all detected markers, re-tag as unsafe, if a certain threshold is passed.

Before rectifying the dataset consisted of about 50% safe labels. This method reduced the number of “safe” tagged vector to about 20% of the data.

Fitting this new retagged dataset to all models as before, and testing again for accuracy, results improvements shown in details reported in Table III. The results improved drastically.

Best classifier selected for the mechanism is SGD (Stochastic Gradient Descend) with loss parameter set to logarithmic. This classifier showed 86% percent accuracy, which could be used with some filtering to suppress false alarm rate even more.

V. SPATIAL RAPID RANDOM TREES

In this section, the Rapid Random Trees (RRT) path planning technique is briefly introduced with spatial extension, which is the basic motion planning drone algorithm. RRT is dealing with high-dimensional spaces by taking into account dynamic and static obstacles including dynamic and non-holonomic robots' constraints.

The main idea is to explore a portion of the space using sampling points in space, by incrementally adding new randomly selected nodes to the current tree's nodes.

RRTs have an (implicit) Voronoi bias that steers them towards yet unexplored regions of the space. However, in case of kinodynamic systems, the imperfection of the underlying metric can compromise such behavior. Typically, the metric relies on the Euclidean distance between points, which does not necessarily reflect the true cost-to-go between states. Finding a good metric is known to be a difficult problem. Simple heuristics can be designed to improve the choice of the tree state to be expanded and to improve the input selection mechanism without redefining a specific metric.

A. *RRT Stages*

The RRT method is a randomized one, typically growing a tree search from the initial configuration to the goal, exploring the search space. These kinds of algorithms consist of three major steps:

1. **Node Selection:** An existing node on the tree is chosen as a location from which to extend a new branch. Selection of the existing node is based on probabilistic criteria such as metric distance.
2. **Node Expansion:** Local planning applied a generating feasible motion primitive from the current node to the next selected local goal node, which can be defined by a variety of characters.
3. **Evaluation:** The possible new branch is evaluated based on cost function criteria and feasible connectivity to existing branches.

These steps are iteratively repeated, commonly until the planner finds feasible trajectory from start to goal configurations, or other convergence criteria.

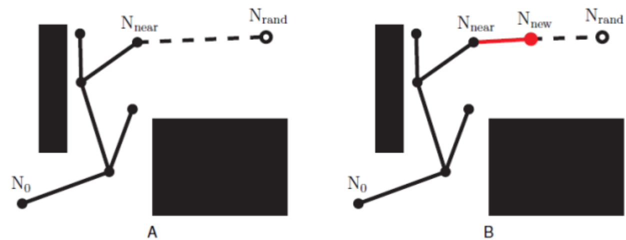


Figure 13. The RRT algorithm: (A) Sampling and node selection steps; (B) Expansion step.

A simple case demonstrating the RRT process is shown in Figure 13. The sampling step selects N_{rand}, and the node selection step chooses the closest node, N_{near}, as shown in Figure 13.A. The expansion step, creating a new branch to a new configuration, N_{new}, is shown in Figure 13.B. An example for growing RRT algorithm is shown in Figure 14.

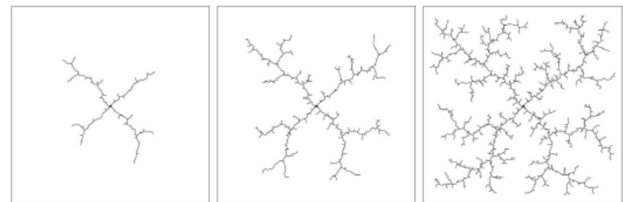


Figure 14. Example for growing RRT algorithm.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

B. Spatial RRT Formulation

We formulate the RRT planner and revise the basic RRT planner for a 3D spatial analysis case for a continuous path from initial state x_{init} to goal state x_{goal}:

1. **State Space:** A topological space, X.
2. **Boundary Values:** $x_{init} \subset X$ and $x_{goal} \subset X$.
3. **Free Space**: A function $D: X \rightarrow \{true, false\}$ that determines whether $x(t) \subset X_{free}$ where X_{free} consist of the attainable states outside the obstacles in a 3D environment.
4. **Inputs:** A set, U, contains the complete set of attainable control efforts u_i, that can affect the state.
5. **Incremental Simulator:** Given a current state, $x(t)$, and input over time interval Δt, compute $x(t+\Delta t)$.
6. **3D Spatial Analysis:** A real value function, $f(x; u, OCP_i)$ which specifies the cost to the center of 3D visibility volumes cluster points (OCP) between a pair of points in X.

C. Spatial RRT Formulation

We present a revised RRT pseudo code described in Table II, for spatial case generating trajectory T, applying K steps from initial state x_{init}. The f function defines the dynamic model and kinematic constraints, $\dot{x} = f(x; u, OCPi)$, where u is the input and OCP_i set the next new state and the feasibility of following the next spatial visibility clustering point.

TABLE II. SPATIAL RRT PSEUDO CODE

```
Generate Spatial RRT (x_init; K; Δt)
T.init (x_init);
For k = 1 to K do
        x_rand ← random.state();
        x_near ← nearest.neighbor (x_rand; T );
        u ← select.input (x_rand; x_near);
        x_new ← new.state (x_near; u; Δt; f);
        T.add.vertex (x_new);
        T.add.edge (x_near; x_new; u);
End
  Return T
```

D. Search Method

Our search is guided by following spatial clustering points based on 3D visible volumes analysis in 3D urban environments, i.e., Optimal Control. The cost function for each next possible node (as the target node) consists of probability to closest OCP, P_{OCPi}, and probability to random point, P_{rand}.

In case of overlap between a selected node and obstacle in the environment, the selected node is discarded, and a new node is selected based on P_{OCPi} and P_{rand}.

E. STP Planner Pseudo-Code

We present our STP planner pseudo code described in Table III, for spatial case generating trajectory T with search space method presented above. The search space is based on P_{OCPi} and P_{rand}. We apply K steps from initial state x_{init}. The f function defines the dynamic model and kinematic constraints, $\dot{x} = f(x; u)$, where u is the input and $OCPi$ are local target points between start to goal states.

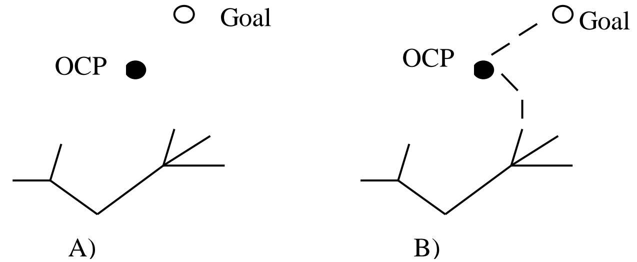


Figure 15. STP Search Method: (A) Start and Goal Points; (B) Explored Space to the Goal Through OCP

F. Completeness

Motion-planning and search algorithms commonly describe 'complete planner' as an algorithm that always provides a path planning from start to goal in bounded time. For random sampling algorithms, 'probabilistic complete planner' is defined as: if a solution exists, the planner will eventually find it by using random sampling. In the same manner, the deterministic sampling method (for example, grid-based search) defines completeness as resolution completeness.

Sampling-based planners, such as the STP planner, do not explicitly construct search space and the space's boundaries, but exploit tests with preventing collision with obstacles and, in our case, taking spatial considerations into account. Similarly, to other common RRT planners, which share similar properties with the STP planner, our planner can be classified as a probabilistic complete one.

VI. SIMULATIONS

The quadcopter we used in this research is a Parrot Bebop2 drone. It is a GPS drone with full HD 1080p wide-angle video camera with 3-axis digital stabilization, that can also take 14MB still pictures.

Bebop2 has GPS guided Return Home feature, strong 6” propellers, long rage communication (with WiFi extender or Skycontroller remote), which makes it suitable for a windy outdoors flight.

The Bebop2 drone uses seven different sensors simultaneously to keep it stable and produce an extremely stabilized video even when the drone makes tiny maneuvers to keep itself in place, the apparent view to the user looks like the drone is in fixed position as if it was hanging on a crane. Also, there are no moving parts when we pan/tilt the camera, it is done entirely by changing the relevant pane in the full fisheye image.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE III. STP PLANNER PSEUDO CODE

```
STP Planner (x_init; x_Goal ;K; Δt; OCP)
T.init (x_init);
x_rand ← random.state();
x_near ← nearest.neighbor(x_rand; T );
u ← select.input(x_rand; x_near);
x_new ← new.state.OCP (OCP_1; u; Δt; f);
While x_new ≠ x_Goal do
        x_rand ← random.state();
        x_near ← nearest.neighbor(x_rand; T );
        u ← select.input(x_rand; x_near);
        x_new ← new.state.OCP (OCP_i; u; Δt; f);
        T.add.vertex(x_new);
        T.add.edge(x_near; x_new; u);
end
return T;

Function new.state.OCP (OCP_i;u; Δt; f)
Set P_OCPi , Set P_rand
p←uniform_rand[0..1]
if   0 < p < P_OCPi
     return x_new = f(OCP_i,u,Δt);
else
     if P_OCPi < p < P_rand+ P_OCPi
then
     return RandomState();
end.
```

Parrot Ground SDK includes software development suite that provides a tool for developers to communicate and control with Parrot drones that uses AR.SDK3 framework, e.g., Mambo, Bebop, Disco, and Anafi. It also includes a simulator platform called Sphinx, built on Gazebo platform, with Parrot drones not just as models but with full featured firmware that are similar to the ones on the equivalent physical drones. This allows developers to fully test and debug their programs with real firmware feedback from a drone in mid-flight without the risk of injury or damages to equipment.

Ground SDK also provides a python wrapper called Olympe, to easily control drone objects. We preferred a third-party implementation named pyparrot, which is better documented and fully open-sourced, so it would be easier to add or change functionality to my needs.

A. ArUco Markers

The first problem we had to deal with, involves detection and identification of the landing pad. Afterword, we had to gather all planar information to pass to the decision mechanism for processing.

In order to simplify detection and get a fast and robust identification and planar information of the target, we used AR-tags on a specially designed landing pad.

Specifically, the use of off-the-shelf open source ArUCO seem to be a simple solution (other implementations of AR-tags, e.g., APRIL-TAGS may be suitable as well).

Implementation of ArUco marker detection exists in open-source library OpenCV, available for c/c++ and python. In order to get the marker real-world coordinates, we need the projection matrix of the camera and the distortion coefficients vector. To get these parameters a calibration is needed to be done once, then it could be loaded through a configuration file. The calibration process also available in OpenCV documentation, using a printed checkboard of known dimensions, and about twenty shots in different orientations and locations across the screen.

We incorporate different marker sizes to be able to detect markers in different distances from the target landing pad and follow the tags. ArUco Markers also have tag ID encoded in them so we even know which tag we are seeing and thus what size it is or where it is located on the board.

TABLE IV. IMPROVEMENTS IN ACCURACY OF CLASSIFIERS

Classifier type	Best accuracy	Before correction	Improve ment
SGD, epsilon insensitive	83.450%	57.341%	26.11%
SGD, hinge	85.062%	75.716%	9.35%
SGD, huber	81.876%	59.841%	22.04%
SGD, log	**86.175%**	**73.658%**	**12.52%**
SGD, modified huber	86.131%	73.362%	12.77%
SGD, squared eps. Insensitive	82.019%	59.6%	22.42%
SGD, squared hinge	85.891%	73.857%	12.03%
SGD, squared loss	82.942%	57.171%	25.77%
Perceptron	85.470%	74.579%	10.89%
Bernoulli NB	81.664%	62.317%	19.35%
Passive Aggressive Classifier	84.041%	74.455%	9.59%

B. Implementation

To get control over a Bebop2 Drone, we found two python wrappers that we could use, and tested both of them. The first one comes with a Parrot Ground-SDK suite which includes the Sphinx Simulator, called Olympe. The Second wrapper pyparrot, originally developed for the Parrot Mambo, but now capable of controlling most of the newer generation Parrot drones.

We decided to work with pyparrot due to two main reasons: 1. Olympe used a closed virtual environment that made it harder to install additional packages using pip. 2. pyparrot is an open source, making it easy to adapt and change to my needs, it also suggests two types of video handling class: the first one uses FFMPEG and the other opens SDP file with VLC on a separate thread. Both methods were slow and missed critical frames especially in SEARCH mode, when the camera rotates to find the target. Sometimes the video smeared so badly we could barely recognize the landing pad even when we knew where it was there.

We changed the video handler to run on a separate thread (like the VLC option on pyparrot) only that in my

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

implementation we used standard OpenCV capturing module VideoCapture to open SDP file (contains IP, port, codec) for streaming coming from the drone or sphinx (depends on DRONE_IP parameter in the code), and another separate thread for the automation state machine that runs the different stages of this autonomous mission control and landing mechanism.

For proof of concept, all experiments were simulated in Gazebo based Sphinx simulator without moving wave simulations, or any automated changes in landing-pad angles or position. The changes were made manually by rotating the pad during simulation when the drone was waiting to get a safe signal from either classifier or calculations.

The experiment also did not simulate the use of "Return Home" functionality and assumed to start near target at about five meters in a random position.

The drone starts to search around to get a visual of the landing pad, then fly to set exactly above while looking directly down (-90 degrees below horizon).

Drone initiates with slow descend while keeping target in the middle of the frame, until reaches height of less than 50cm.

In this stage, decision mechanism under test should trigger "safe" when ArUco markers of the pad will be in a position that is regarded flat enough to be considered as safe.

In a preliminary experiment, we found that the classifier that we trained, could not get to a "safe" decision even when the landing pad was flat without any movements. Same classifier was tested with images from web-cam input seems to work fine, this could be issue caused by miscalibration of the camera. These inaccuracies cause ArUco functions that heavily relay on camera calibration, to produce different range of data relative to what the classifier was trained with (data from an OpenGL graphics drawn landing pad). This method should be further explored in future work.

Simplified manual calculation that work directly on data from ArUco functions output, could also be easily recalibrated and adjustable to fit with data ranges of mis-calibrated data. Finally, running full scenario of the experiment with landing pad on unsafe initial position got the drone flying above it and waiting, then manually flatten the landing pad, made the decision mechanism to trigger "safe" and send a landing command to the drone, which landed in the desired spot.

VII. Conclusion and Future Work

In this work we introduced a mechanism for autonomous landing a quadcopter in. The work focused to assist in the final stage of an autonomous mission, when drone returned to home, but still needs to find exact position of landing on the target and dealing with sea-level motion of the target.

In this study we developed a training simulator to create large data set of visual input, produced by OpenGL graphics in a controllable manner.

Also, we compared different types of trained classifiers to find best match to our particular data, and competed best classifier vs. direct observation and improvements as can be seen in Table IV.

For conclusion, the ArUco functions produce enough information regarding marker positions to be used manually and get a satisfying result for that manner. It is fast and robust and easily read to get a quick answer to whether it is safe or not, and the use of a classifier is not necessary.

References

[1] O. Gal and Y. Doytsher, "Autonomous Drone Landing in 3D Urban Environment Using Real-Time Visibility Analysis," GEOProcessing 2023, The Fifteenth International Conference on Advanced Geographic Information Systems, Applications, and Services, pp. 67- 72, 2023.

[2] Parrot Inc., "developer.parrot.com," 2020. [Online]. Available: https://developer.parrot.com/docs/olympe/userguide.html.

[3] A. McGovern, "pyparrot github repository," Jan. 2020. [Online] https://github.com/amymcgovern/pyparrot.

[4] OpenCV, Open Source Computer Vision Library, 2015.

[5] K. Williams, "A Summary of Unmanned Aircraft Accident/Incident Data: Human Factors Implications," The Federal Aviation Admistrator Oklahoma City, 2004.

[6] A. F. Cobo and F. C. Benıtez, "Approach for Autonomous Landing on Moving Platforms based on computer vision," The International Journal of Computer Vision, vol 4., 2016.

[7] L. Daewon, R. Tyler, and K. H. Jin, "Autonomous landing of a VTOL UAV on a moving platform using image-based visual servoing," IEEE International Conference on Robotics and Automation, pp. 971-976, 2012.

[8] T. Merz, S. Duranti, and G. Conte, "Autonomous Landing of an Unmanned Helicopter Based on Vision and Inertial Sensing," Experimental Robotics IX, pp. 343-352, 2006.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org