2017 International Conference on
Unmanned Aircraft Systems (ICUAS)
June 13-16, 2017, Miami, FL, USA

Specification and Evaluation of Geofence Boundary Violation Detection Algorithms*

Mia N. Stevens[1], Hossein Rastgoftar[2], and Ella M. Atkins[3]

Abstract— **This paper studies two methods of geofence boundary violation detection. The first method is *Ray Casting*, which iterates over each geofence boundary edge to determine if a given position of interest is inside the geofence. The second method, called *Triangle Weight Characterization* (TWC), subdivides the geofence domain into a finite number of triangles, then iterates over each triangle to determine if the given position of interest is inside the geofence. We apply the TWC and Ray Casting methods to case studies that include both keep-in and keep-out geofence boundaries.**

I. INTRODUCTION

Unmanned aircraft systems (UAS) continue to proliferate and now can be operated commercially with little overhead through the FAA's Part 107 rules [1]. UAS applications range from last-mile package deliveries to agricultural and infrastructure inspection to disaster relief support, not to mention hobbyist flights. A micro-scale UAS or micro-air vehicle (MAV) may pose little risk to people or property, but such a vehicle has limited range and cannot carry payload beyond a small camera. Even small UAS can pose a safety risk through both fast-spinning propellers and impact. NASA is working with industry and academic partners to develop a UAS Traffic Management (UTM) system of which a key component is electronic geofencing [2].

Geofences assign each UAS an empty flight volume in which they are authorized to operate. The geofence can also be used as a mechanism to assure a low-flying UAS only operates low over a property with landowner permission. A geofence can be classified as a keep-in (inclusion) geofence or a keep-out (exclusion) geofence. The keep-in geofence defines a bounded flight volume for the UAS, while the keep-out geofence defines general volumes to avoid as well as cut-outs within a keep-in geofence. A keep-out geofence marks a no fly zone for the UAS. Public properties such as national monuments and private properties such as a backyard pool may be protected by low-altitude keep-out geofencing.

Given defined geofence boundaries, the geofence system consists of two logic units: the detection of geofence violations and the response to a geofence violation. There are many possible responses to a geofence violation including but not limited to alerting the pilot, cutting the aircraft power, or an alternative guidance scheme designed to respect the geofence boundaries [3]. This paper focuses on the detection of geofence violations through the application of two algorithms: Ray Casting [4]–[6] and Triangle Weight Characterization (TWC) [7]–[9].

Section II introduces the geofence violation detection framework and the algorithms being considered. Sections III and IV discuss the details of the Ray Casting and Triangle Weight Characterization (TWC) algorithms respectively. Section V presents a series of geofencing case studies to compare the geofence boundary violation detection algorithms. Section VI discusses areas for future work including the usage of geofencing in urban areas and the incorporation of UAS into the existing airspace, and Section VII presents conclusions.

II. BACKGROUND

Assumptions made in this work are listed in Tables I and II. An assumption of constant flight altitude enables this paper to focus on algorithms that detect lateral geofence boundary violations. Note that these algorithms will be extended in future work to remove the constant altitude assumption.

TABLE I: Assumptions of each geofence violation detection system.

Assumption
Geofence boundaries are drawn conservatively such that there is sufficient time to detect and react to a boundary crossing
Geofence boundaries remain unchanged for the duration of a flight
Geofences contain vertical and lateral boundaries to define the airspace available to the UAS
Vertical geofence boundaries (altitude ceiling and floor) are constant above ground level (AGL) or mean sea level (MSL)
Flights occur at constant altitude so only lateral boundaries need to be considered
Each geofence consists of exactly one keep-in geofence and any number of keep-out geofences

TABLE II: Assumptions of each keep-in and keep-out geofence.

Assumption
The geofence boundary is not self-intersecting as shown in Figure 1
The geofence boundary is specified as a list of vertices, e.g., GPS Latitude/Longitude or relative position points, in clockwise or counterclockwise order

A geofence violation occurs when the UAS is outside the keep-in geofence or inside a keep-out geofence. The flow

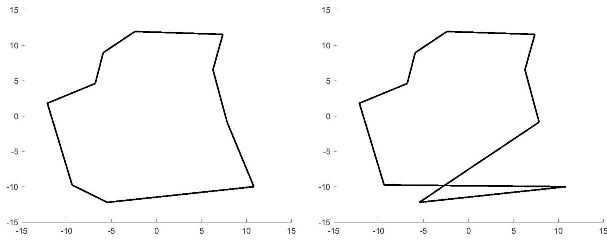
*This work was supported in part by a subcontract from Soar Technology, Inc. under Phase II SBIR Contract No. FA8650-15-C-2629.

[1]Mia N. Stevens is a Robotics Institute PhD Candidate, University of Michigan, Ann Arbor, MI 48109, USA minist@umich.edu

[2]Hossein Rastgoftar is an Aerospace Engineering Postdoctoral Scholar, University of Michigan, Ann Arbor, MI 48109, USA hosseinr@umich.edu

[3]Ella M. Atkins is an Aerospace Engineering Professor, University of Michigan, Ann Arbor, MI 48109, USA ematkins@umich.edu

978-1-5090-4494-8/17/$31.00 ©2017 IEEE



(a) Acceptable lateral geofence boundary.

(b) Unacceptable lateral geofence boundary.

Fig. 1: Examples of valid and invalid (unacceptable) lateral geofence boundaries using the same vertex list.

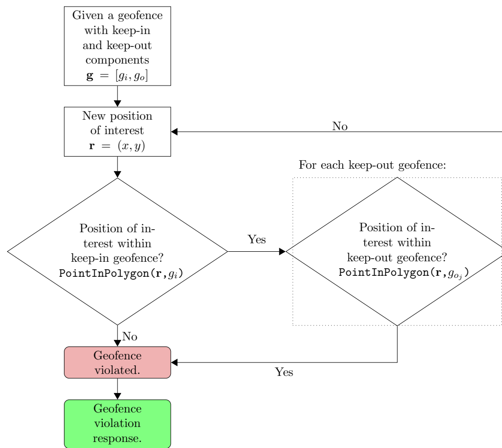


Fig. 2: General geofence violation detection algorithm for a single keep-in geofence and a known number of keep-out geofences.

Algorithm 1 Geofence Boundary Violation Detection

Input: $\mathbf{r}$ is the position of interest
$\mathbf{g} = [g_i, g_o]$ g_i is the keep-in geofence, g_o is the list of keep-out geofences
Output: **true** if $\mathbf{r}$ does not violate $\mathbf{g}$, otherwise **false**

1: **if not** `PointInPolygon`$(\mathbf{r}, g_i)$ **then**
2:     **return false**
3: **end if**
4: **for all** g_{o_j} in g_o **do**
5:     **if** `PointInPolygon`$(\mathbf{r}, g_{o_j})$ **then**
6:         **return false**
7:     **end if**
8: **end for**
9: **return true**

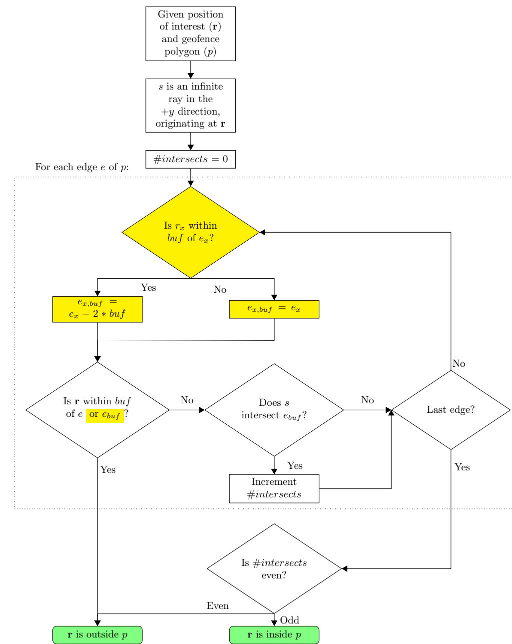


Fig. 3: Ray Casting algorithm. This algorithm becomes Fast Ray Casting when the yellow highlighted sections are excluded.

chart in Figure 2 shows the basic procedure of assessing whether or not a geofence violation has occurred, which is also presented in Algorithm 1. The input parameters are $\mathbf{r}$ and $\mathbf{g}$. $\mathbf{r} = (x, y)$ is the current UAS lateral plane position to check for geofence violation. The geofence is specified by $\mathbf{g} = [g_i, g_o]$ where g_i is the keep-in geofence boundary polygon and $g_o = \{g_{o_1}, ..., g_{o_n}\}$ is the set of keep-out boundaries. g_{o_j} is the jth of n keep-out geofence boundary polygons. The `PointInPolygon()` function is either the Ray Casting algorithm or the TWC algorithm. The loop is executed for every new state estimate, as long as a geofence violation has not occurred. A detected geofence violation causes the geofence system to transition from the violation detection subsystem to the violation response subsystem, which is not addressed in this work.

The detection of a lateral geofence violation can be considered an application of the point-in-polygon problem, which is commonly discussed in the field of computer graphics. The algorithms presented in this work, Ray Casting (see Figure 3) and Triangle Weight Characterization (see Figure 4), are two solutions to this problem. Ray Casting projects an infinite ray from the position of interest then loops over each edge of the considered polygon to determine if an even or odd number of edges are intersected. Zero or an even number of intersects indicates that the position of interest is outside the polygon; an odd number of indicates that the position of interest is within the polygon. TWC divides the polygon into triangles during an initialization procedure that is executed once per flight. Then, for each position of interest, TWC loops over each triangle of the polygon to determine if the position of interest is contained within that triangle. If the position of interest is within a triangle, then, it is within the polygon and the loop terminates; otherwise the position of interest is outside each triangle and outside the polygon.

III. Ray Casting

The Ray Casting algorithm determines whether or not the position of interest, $\mathbf{r}$, is inside a given polygon, p,

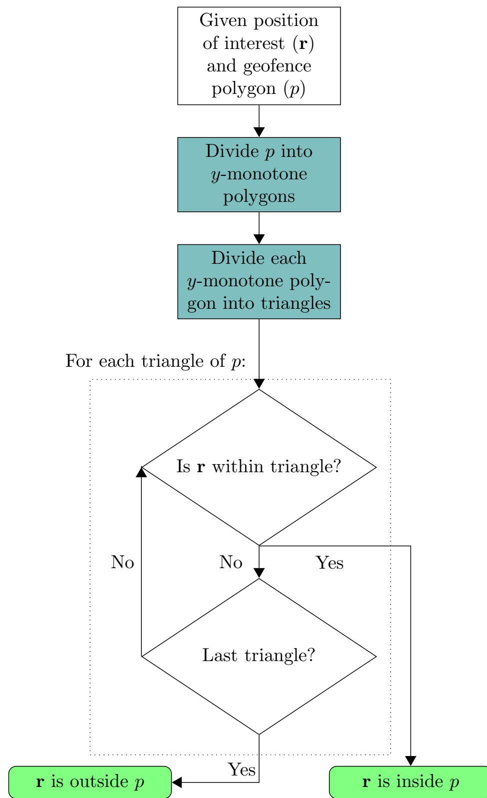


Fig. 4: Triangle Weight Characterization algorithm. Blue highlight indicates initialization steps that are only executed once per flight.

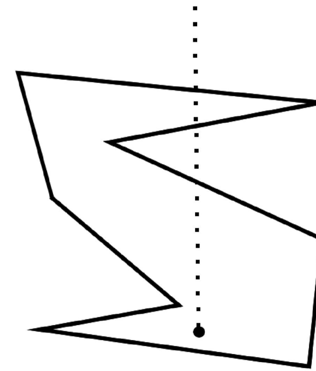


Fig. 5: Ray Casting [4].

Algorithm 2 `PointInPolygon()` - Ray Casting

Input: p is a simple polygon
$\mathbf{r}$ is the position of interest
buf is a buffer distance
Output: **true** if p contains $\mathbf{r}$, otherwise **false**
1: $count = 0$
2: s is an infinite ray in the $+y$ direction, originating at $\mathbf{r}$
3: **for all** edges e in p **do**
4: **if** $\mathbf{r}_x$ is within buf of e_x **then**
5: $e_{x,buf} = e_x - 2 * buf$
6: **else**
7: $e_{buf} = e$
8: **end if**
9: **if** $\mathbf{r}$ is within buf of e or e_{buf} **then**
10: **return false**
11: **end if**
12: **if** s intersects e_{buf} **then**
13: $count = count + 1$
14: **end if**
15: **end for**
16: **if** $count$ is odd **then**
17: **return true**
18: **else**
19: **return false**
20: **end if**

by projecting an infinite ray from $\mathbf{r}$ (see Figure 5). If the infinite ray intersects an odd number of polygon edges, then $\mathbf{r}$ is contained in p, otherwise, $\mathbf{r}$ is outside of p. A basic outline of the Ray Casting algorithm is shown in Algorithm 2. The algorithm is based on the formulation presented by Narkawicz and Hagen [4]. Because the Ray Casting algorithm iterates over all edges of p and does not have an initialization step, if the geofence boundaries change from one time step to the next, code execution and results of the Ray Casting algorithm are not impacted.

In this implementation, the ray is cast in the positive y-direction, as seen in Figure 5. To prevent the infinite ray from intersecting a corner of the polygon, a buffer distance, buf, is defined. Any corner of p with an x-value within buf of $\mathbf{r}_x$ is perturbed by $-buf * 2$ along the x-axis. An alternative to perturbing the vertices that coincide with the $\mathbf{r}_x$ would be to count each intersection of that vertex as $count = count + 1/2$ instead of $count = count + 1$ [5].

Lines $9-11$ of Algorithm 2 (and highlighted in yellow in Figure 3) state that if the position of interest, $\mathbf{r}$, is within the buffer distance, buf, of the edge currently being considered, then $\mathbf{r}$ is considered outside polygon p. This is an important check to run because buf could be defined to account for uncertainty that might exist in the state estimate of the UAS. These lines are highlighted because their inclusion (denoted Ray Casting) or exclusion (denoted Fast Ray Casting) significantly impact the runtime of the algorithm, as shown below in Section V.

IV. Triangle Weight Characterization (TWC)

The second boundary violation detection algorithm, Triangle Weight Characterization, consists of an initialization step and a run-time step as shown in Algorithm 3. The initialization step must be executed for all keep-in and keep-out geofences when the system first activates. If there are any changes to any of the geofence boundaries after the original initialization, each keep-in or keep-out geofence that is changed must be initialized again. The initialization step subdivides each of the original geofences from simple polygons to y-monotone polygons [9] and then to triangles

[8]. The run-time step checks whether the position of interest is within each triangle. If the position of interest is inside any of the triangles, then it is within that polygon. Otherwise, it is outside the polygon.

Algorithm 3 `PointInPolygon()` - Triangle Weight Characterization

Input: p is a simple polygon
  $\mathbf{r}$ is the position of interest
Output: **true** if p contains $\mathbf{r}$, otherwise **false**
  `Initialization:`
1: Divide p to m y-monotone polygons
2: **for all** y-monotone polygons M in p **do**
3:   Divide polygon M to n triangles
4: **end for**
  `Run-Time:`
5: **for all** N triangles in p **do**
6:   **if** N contains $\mathbf{r}$ **then**
7:     **return true**
8:   **end if**
9: **end for**
10: **return false**

A. TWC Initialization

To divide an arbitrary geofence boundary into non-intersecting triangles, we implement the triangulation method described in Garey et al. [8] which relies on the regularization algorithm presented by Lee and Preparata [9]. To visualize the subdivision of an arbitrary polygon, TWC is applied to the polygon shown in Figure 6. TWC initialization consists of two steps: divide the polygon into monotone polygons [9], and subdivide each monotone polygon into triangles [8]. Each of these steps is executed with respect to the y-axis but would also work if applied to the x-axis.

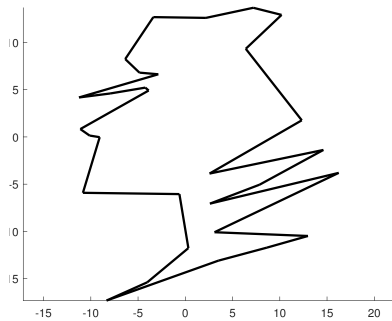


Fig. 6: Randomly generated example geofence. Black - keep-in geofence.

1) Polygon to Monotone Polygons: A y-monotone polygon is defined as a polygon for which all lines parallel to the x-axis intersect a maximum of two edges of the polygon. In order to divide a polygon into monotone polygons, we iterate through the vertices from highest to lowest y-position, then from lowest to highest y-position, adding edges between vertices to create monotone polygons. Vertices with equivalent y-values are iterated over from left to right. [9] The original algorithm creates new edges between existing vertices both inside and outside the original polygon, but geofencing is only interested in the area of the original polygon, so edges added that are outside the original polygon are ignored. An edge is determined to be outside the original polygon when the order of the edge vertices of the newly defined polygon is opposite the order of the original polygon vertices, i.e., clockwise versus counterclockwise. Because some of the newly-generated edges are ignored, this algorithm is executed for each newly created polygon until no new edges are added. This ensures that the polygons being returned are all y-monotone. In Figure 7, each y-monotone polygon is shaded a different color; the original polygon has been divided into five y-monotone polygons.

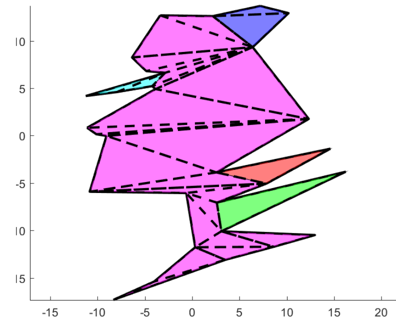


Fig. 7: Randomly generated example geofence. Each background color indicates a separate y-monotone polygon. Dotted lines indicate triangle boundaries. Solid lines indicate the original polygon boundaries.

2) Monotone Polygon to Triangles: For each monotone polygon, the vertices are iterated over from highest to lowest y-value, iteratively adding edges to create triangles. Because the polygons are already y-monotone, all created edges are inside the polygon and therefore kept. For the example geofence, this algorithm is run five times, once for each monotone polygon. Figure 7 illustrates the TWC triangles by dotted lines.

B. TWC at Run-Time

To determine if a position of interest, $\mathbf{r}$, is inside the geofence polygon, we check if $\mathbf{r}$ is inside each triangle. Let vertices of the ith triangular cell be located at $\mathbf{r}_{i_1} = (x_{i_1}, y_{i_1})$, $\mathbf{r}_{i_2} = (x_{i_2}, y_{i_2})$, and $\mathbf{r}_{i_3} = (x_{i_3}, y_{i_3})$. Because a triangle is a $2-D$ convex hull, positions of the ith triangular cell satisfy the following rank condition:

$$\text{Rank}\begin{bmatrix}\mathbf{r}_{i_2}-\mathbf{r}_{i_1} & \mathbf{r}_{i_3}-\mathbf{r}_{i_1}\end{bmatrix} = \begin{bmatrix} x_{i_2}-x_{i_1} & x_{i_3}-x_{i_1} \\ y_{i_2}-y_{i_1} & y_{i_3}-y_{i_1}\end{bmatrix} = 2. \quad (1)$$

Therefore, position of an arbitrary point $\mathbf{r} = (x, y)$ in the motion plane can be uniquely expanded as

$$\begin{aligned} \mathbf{r} =& \mathbf{r}_{i_1} + w_{i_2}\left(\mathbf{r}_{i_2}-\mathbf{r}_{i_1}\right) + w_{i_3}\left(\mathbf{r}_{i_3}-\mathbf{r}_{i_1}\right) \\ =& \left(1-w_{i_2}-w_{i_3}\right)\mathbf{r}_{i_1} + w_{i_2}\mathbf{r}_{i_2} + w_{i_3}\mathbf{r}_{i_3}. \end{aligned} \quad (2)$$

Setting $w_{i_1} = \left(1 - w_{i_2} - w_{i_3}\right)$, Eq. (2) can be rewritten as

$$\mathbf{r} = \sum_{k=1}^{3} w_{i_k} \mathbf{r}_{i_k} \tag{3}$$

where

$$\sum_{k=1}^{3} w_{i_k} = 1. \tag{4}$$

Considering Eqs. (3) and (4), distance weights w_{i_1}, w_{i_2}, and w_{i_3} are obtained from

$$\begin{bmatrix} x_{i_1} & x_{i_2} & x_{i_3} \\ y_{i_1} & y_{i_2} & y_{i_3} \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_{i_1} \\ w_{i_2} \\ w_{i_3} \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5}$$

The distance weights satisfying (5) can expressed as follows:

$$\begin{aligned} w_{i_1}(x,y) &= \frac{\left(x_{i_3} - x_{i_2}\right)\left(y - y_{i_2}\right) - \left(y_{i_3} - y_{i_2}\right)\left(x - x_{i_2}\right)}{\left(x_{i_3} - x_{i_2}\right)\left(y_{i_3} - y_{i_2}\right) - \left(y_{i_3} - y_{i_2}\right)\left(x_{i_3} - x_{i_2}\right)} \\ w_{i_2}(x,y) &= \frac{\left(x_{i_1} - x_{i_3}\right)\left(y - y_{i_3}\right) - \left(y_{i_1} - y_{i_3}\right)\left(x - x_{i_3}\right)}{\left(x_{i_1} - x_{i_3}\right)\left(y_{i_1} - y_{i_3}\right) - \left(y_{i_1} - y_{i_3}\right)\left(x_{i_1} - x_{i_3}\right)}. \\ w_{i_3}(x,y) &= \frac{\left(x_{i_2} - x_{i_1}\right)\left(y - y_{i_1}\right) - \left(y_{i_2} - y_{i_1}\right)\left(x - x_{i_1}\right)}{\left(x_{i_2} - x_{i_1}\right)\left(y_{i_2} - y_{i_1}\right) - \left(y_{i_2} - y_{i_1}\right)\left(x_{i_2} - x_{i_1}\right)} \end{aligned} \tag{6}$$

$w_{i_k}(x,y) = \text{c}$ ($k = 1,2,3$ and c is a constant.) is a line parallel to a triangle side not passing through i_k. As examples, $w_{i_1} = \text{c}$ is a line parallel to triangle side $i_2 - i_3$, $w_{i_2} = \text{c}$ is a line parallel to triangle side $i_3 - i_1$, and $w_{i_3} = \text{c}$ is a line parallel to triangle side $i_1 - i_2$. Also, $w_{i_k}(x_{i_j}, y_{i_j}) = \delta_{k,j}$, where $\delta_{k,j}$ is the Kronecker delta defined as follows:

$$\delta_{k,j} = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases}. \tag{7}$$

In Fig. 8, the $x - y$ motion plane can be divided into seven sub-regions based on the signs of distance weights w_{i_1}, w_{i_2}, w_{i_3}. As seen, distance weights are all positive inside the ith triangular cell.

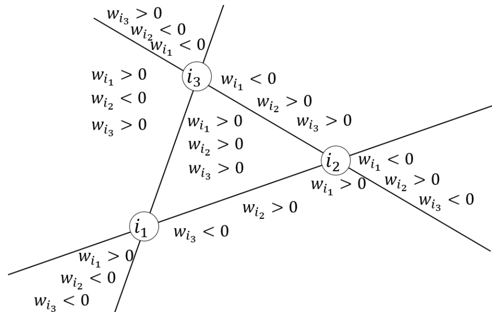


Fig. 8: Division of the motion plane into seven sub-regions based on the signs of distance weights w_{i_1}, w_{i_2}, and w_{i_3}

If the distance weights of one of the triangles are all positive for the position of interest, then the position of interest is within the polygon.

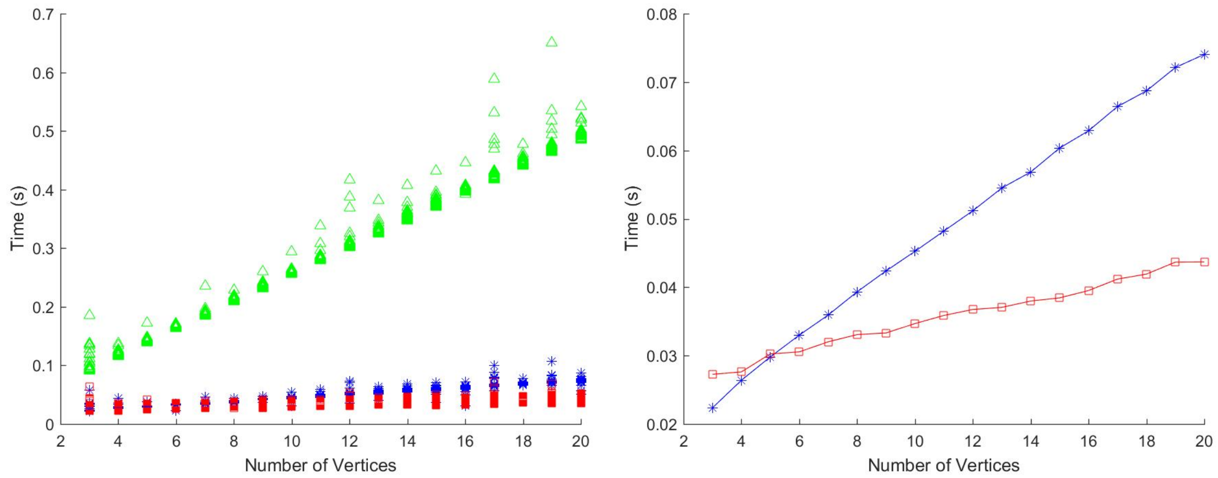
Remark: If a geofence area is sufficiently large, it may not be approximated by a planar surface. For this case, the geofence domain can be considered as a spherical surface with longitude ϕ and longitude λ. The proposed TWC method can still be applied for boundary violation checking over a spherical surface. By substituting x, x_{i_1}, x_{i_2}, x_{i_3}, y, y_{i_1}, y_{i_2}, y_{i_3} by ϕ, ϕ_{i_1}, ϕ_{i_2}, ϕ_{i_3}, λ, λ_{i_1}, λ_{i_2}, λ_{i_3}, weights $w_{i_1}(\phi,\lambda)$, $w_{i_2}(\phi,\lambda)$, and $w_{i_3}(\phi,\lambda)$ can be obtained from Eq. (6). Similar to the planar domain, the UAS is enclosed by the ith sector over the spherical geofence surface, if $w_{i_1}(\phi,\lambda)$, $w_{i_2}(\phi,\lambda)$, and $w_{i_3}(\phi,\lambda)$ are all positive.

V. Case Studies

To evaluate the execution time of the Ray Casting, Fast Ray Casting, and TWC algorithms, we generated 1000 random positions of interest and recorded the total time required to run each geofence violation detection algorithm for each position of interest. These tests were done for 50 randomly generated geofences with 3 to 20 vertices, a total of 900 unique geofences. These geofences represent cases where there is a defined keep-in geofence with zero keep-out geofences. The results are shown in Figure 9. Figure 9(a) presents the time required to evaluate all 1000 positions of interest for each geofence, grouped by number of vertices, and colored by algorithm. Figure 9(b) plots the median total time of geofences with each number of vertices for Fast Ray Casting and TWC. The presented plots do not include the time required for the TWC initialization. The initialization time can be ignored in this analysis because we are assuming that the geofence boundaries are not changed or updated during a given flight, so the initialization of TWC can occur prior to the UAS takeoff.

A. Cases

We executed the Ray Casting, Fast Ray Casting, and TWC boundary check algorithms over three specific geofences: a randomly generated boundary, a path confined to a specific region over the Hudson river in the New York City region, and a simplified geofence boundary over this same region. The map-based geofence case studies were generated using Google My Maps by selecting the latitude and longitude coordinates of the vertices of the keep-in and keep-out geofences as *shapes*. In addition to the geofence boundary vertices, a *Home* position was created inside each of the keep-in geofences to be used as the local origin. The geofence case study maps were downloaded as `.kml` files and imported into MATLAB using the add-on function `kml2struct()`, which generates structures for each *shape* with arrays of the latitude and longitude of each vertex [10]. These arrays of geofence vertices and the local origin were then converted into meters using the MATLAB add-on function `deg2utm()`, which converts vectors of latitude and longitude into UTM coordinates (WGS84) [11]. Once all map features are expressed in meters, the geofence vertices are redefined relative to local origin. The map-defined geofence boundaries use the same functions as the randomly-generated examples.



(a) Execution times for each geofence boundary (50 per vertex count). Ray Casting - green triangles. Fast Ray Casting - red circles. Triangle Weight Characterization - blue stars.

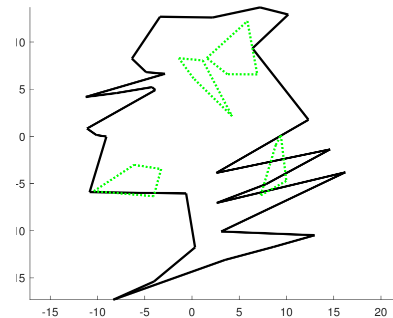
(b) Median execution times for each geofence boundary vertex count. Fast Ray Casting - red circles. Triangle Weight Characterization - blue stars.

Fig. 9: Summed execution time for geofence violation check of 1000 positions of interest as a function of number of geofence boundary vertices ($3-20$ vertices), 50 unique geofence boundaries generated for each.

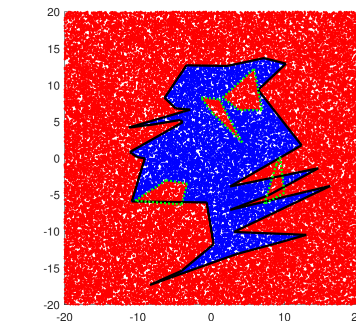
To test the boundary check algorithm using a randomly-generated geofence boundary consisting of a keep-in geofence with 30 vertices and four keep-out geofences with 4 vertices each, we use the MATLAB command `rand()` to generate $50,000$ random (x, y) positions of interest. The random positions of interest are then tested for geofence boundary violation. The results are displayed in Figure 10. The positions determined to be outside the boundaries are shown in red, and the positions judged to be inside the boundaries are shown in blue. The absence of red points within the boundaries and of blue points outside the boundaries indicates that the boundary violation detection algorithm is functioning properly.

The first case study defines geofence boundaries as shown in Figure 11(a). This scenario simulates a case where a UAS has been given permission to fly over the river near a populated area but is not allowed to fly over the surrounding cities or islands. It is represented as a keep-in geofence with 17 vertices and three keep-out geofences with 6, 6, and 9 vertices respectively. Additional keep-out geofences could be defined to prevent the UAS from flying over ships traveling along the river. This scenario mimics what will likely be a common model where large UAS are confined to flying at higher altitudes shared by manned aircraft each following a preplanned and approved flight path, or at low altitudes shared with other unmanned aircraft potentially with no preplanned flight path. In this scenario, the geofence system acts to allow the UAS to fly its desired route while also preventing the UAS from straying too close to populated regions.

The second case study defines geofence boundaries as shown in Figure 12(a). This setup simulates a case where a UAS has been given permission to fly over all areas within a geofence area with the exception of sensitive (keep-out)



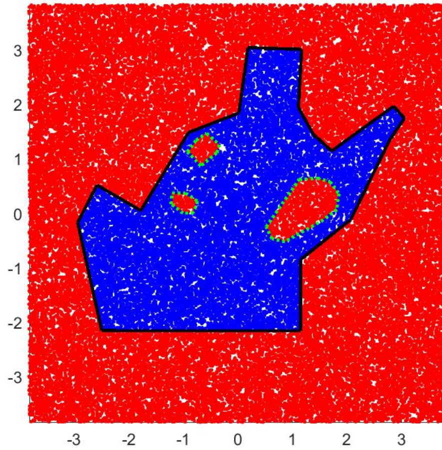
(a) Black - keep-in geofence. Green - keep-out geofences.



(b) $50,000$ randomly generated positions of interest. Blue - within geofence. Red - violating geofence.

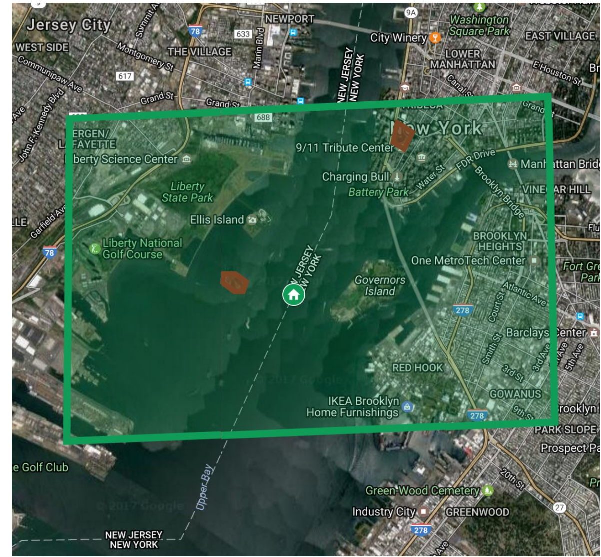
Fig. 10: Randomly generate example geofence.

(a) Home icon - local origin. Green - keep-in geofence. Red - keep-out geofences.

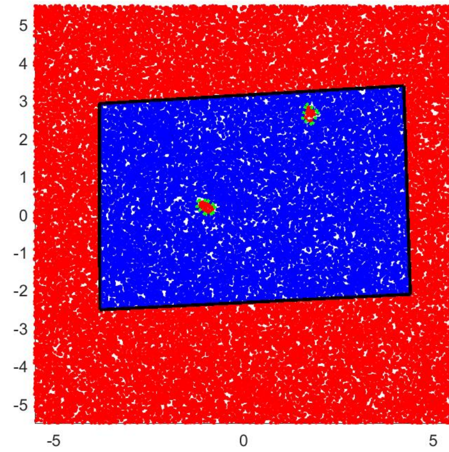


(b) 50, 000 randomly generated positions of interest (in kilometers). Blue - within geofence. Red - violating geofence.

Fig. 11: Case Study 1 - River flight near urban area.



(a) Home icon - local origin. Green - keep-in geofence. Red - keep-out geofences.



(b) 50, 000 randomly generated positions of interest (in kilometers). Blue - authorized flight location. Red - location violating a geofence boundary.

Fig. 12: Case Study 2 - Urban area flight.

regions surrounding the World Trade Center Memorial and the Statue of Liberty. It is represented as a keep-in geofence with 4 vertices and two keep-out geofences with 4 and 6 vertices respectively. Note that this work presumes the existence of a separate detect-and-avoid system for terrain, buildings, and other aircraft enabling geofence boundaries to be defined by airspace permissions not obstacles, e.g., tall Manhattan structures. Indeed, the UAS may acquire permission to fly between buildings and over populated areas given mission need. In this scenario, the geofence system serves to prevent the UAS from straying too far from its operator or into an airspace sector the UAS does not have permission to occupy given (future) beyond line-of-sight (BLOS) operational approval.

B. *Evaluation of Case Studies*

For each of these cases, Ray Casting, Fast Ray Casting, and TWC algorithms were applied to 50, 000 randomly generated nearby positions of interest. The time (in seconds) required to run all 50, 000 points for each of the methods is presented in Table III and Figure 13. These run times are for a MATLAB implementation on a Windows laptop; an implementation in a compiled language such as C would execute more rapidly but with comparable relative factors. In order to minimize the impact of any background processes on the execution times, the MATLAB execution script interspersed the execution of each of the three algorithms being tested.

Based on the results presented in Figure 9 and Table III, it would be easy to conclude that either Fast Ray Casting or TWC were good choices for the geofence violation

TABLE III: Case Study Results Table - run time in seconds for $50,000$ randomly generated points of interest

	Random	Case 1	Case 2
Ray Casting	65.31 s	54.12 s	23.33 s
Fast Ray Casting	9.46 s	7.25 s	5.36 s
(TWC Initialization) Triangle Weight Characterization	(0.25 s) 9.19 s	(0.23 s) 8.32 s	(0.16 s) 4.12 s

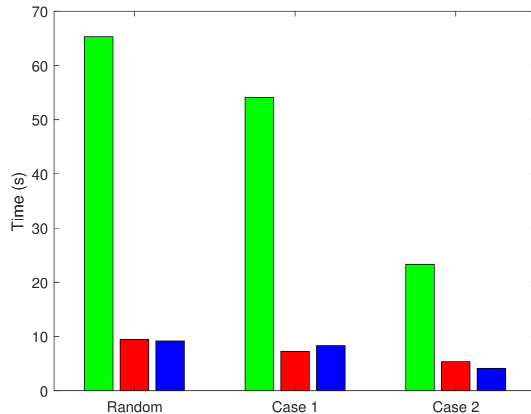


Fig. 13: Run time for $50,000$ positions of interest. Ray Casting - green. Fast Ray Casting - red. Triangle Weight Characterization - blue.

algorithm, and that Fast Ray Casting algorithm would be the best choice since it does not require an initialization step, unlike TWC. However, neither of those algorithms include the buffer distance that is incorporated into Ray Casting, which allows for state estimation error bias. As such, Ray Casting could provide value that justifies the additional overhead. In future work TWC could be augmented to take the buffer distance check into account, providing another comparison point between the two algorithms.

VI. Discussion

The difference between Ray Casting and Fast Ray Casting is that Ray Casting includes a check of the position of interest's proximity to the geofence boundaries. The edge proximity check is a useful feature because it enables to usage of a buffer distance to allow for state estimation imprecision. Further work is required to extend the current TWC algorithm to include an edge proximity check similar to the Ray Casting algorithm. This extended TWC algorithm is expected to run more quickly than the Ray Casting algorithm because it will only need to check the proximity of the position of interest for triangles near geofence edges.

Another direction for future work is to use the geofence triangulation from TWC to plan UAS flight paths that do not violate the geofence. For a single keep-in geofence with no keep-out geofence regions, if the UAS is inside the geofence, the triangle of the geofence that contains the UAS is known. To plan a path to another location within the geofence without violating the geofence boundaries, we can exploit the fact that for geofences with more than three vertices, each triangular subdivision shares an edge with at least one other (adjacent) triangle. Given the current UAS position $\mathbf{r}(t_k) = (x(t_k), y(t_k))$, the UAS path planner can choose $\mathbf{r}(t_{k+1})$ to be within an adjacent triangle and plan to move on the line segment connecting $\mathbf{r}(t_k)$ and $\mathbf{r}(t_{k+1})$. By the nature of TWC, any straight line path within a triangle will not violate the geofence boundaries. Similarly, every point along the straight line path from one triangle into an adjacent triangle will not contain a geofence boundary violation. As a result, any path that begins and ends within a keep-in geofence can be generated by planning only straight line paths between waypoints in the same triangle or in adjacent triangles.

For cases where the flight path has already been planned, the geofence boundary violation checks can be used to validate the flight path prior to takeoff by sampling points along the flight path and ensuring that no point violates the geofence boundaries. The sampling of the path needs to be fine enough that likelihood of a geofence violation occurring between sample points is negligible. The time required to complete this check is not a mission critical factor because this check is executed prior to takeoff. If a geofence boundary violation is detected in the planned flight path either the flight path must be updated to avoid the violation or a modification to the geofence boundaries must be requested to enable the desired flight path. Once the planned flight path is declared free of geofence boundary violations, the only violation risks during flight are from system failure, environmental factors, or changes to the geofence due to emergency flight vehicles.

This work focused on the evaluation of static geofence boundaries where the geofence boundaries either never change or are only analyzed for a single time step. Geofence boundaries could also change over time, moving from static to dynamic geofencing. When the geofence boundary changes, there are three possible actions that the geofence system can take to incorporate updated information:

1) Complete the current iteration of Algorithm 1 with the old geofence boundaries and incorporate the new geofence boundaries in the next iteration,
2) Complete the current iteration of Algorithm 1 with the new geofence boundaries and use the new geofence boundaries in the next iteration,
3) Halt the current iteration of Algorithm 1 and immediately begin a new iteration with the new geofence boundaries.

Actions 1 and 2 cause a maximum of one iteration using incorrect geofence boundaries. Action 3 is unique in that it does not use any incorrect data, but may cause problems if the geofence / UAS system is not robust to timing inconsistencies in the geofence boundary violation detection algorithm. Dynamic geofences are important for the incorporation of UAS into the manned airspace because they enable changes to geofence boundaries in emergency situations such as when a law enforcement vehicle or a medical helicopter requires the airspace previously allocated to the UAS geofence. The proposed approach is sufficiently

fast that it is directly applicable to dynamic geofencing with a high-frequency (multi-Hz) update, so typically any of the presented options for handling geofence boundary updates will be sufficient.

Being able to handle geofence boundary updates is particularly important in crowded airspace, such as urban regions, where other aircraft could be incorporated into the geofence system as additional keep-out geofence regions. This would allow multiple UAS to operate in the same or overlapping keep-in geofences without fear of a midair collision, as long as the position of each UAS is broadcast at a known regular interval. This setup does not protect against uncooperative UAS or systems that are unable to broadcast their positions such as kites or simple RC aircraft. A separate sense and avoid system is needed to avoid unknown obstacles.

The inclusion of sense and avoid system in addition to a geofencing system is critical to safe flight. In an urban environment, there are obstacles to be avoided other than other aircraft, including buildings, power lines, and street lights. These are objects that are known and constant, but to exclude each individual object would be impractical. The issue is further complicated by the existence of urban canyons, where GPS is denied due to the surrounding buildings. In these areas, even if every obstacle were designated a keep-out geofence, accumulated state estimation error might make it impossible for the geofence to guarantee a collision free flight. In these cases, the addition of a sense and avoid system could enable safe flight through a keep-in geofence region without relying on the specification of every obstacle as a keep-out region.

When operating at higher altitudes, in shared airspace with manned aircraft, large UAS can still benefit from geofencing. The keep-in geofence ensures that the UAS does not exit its designated flight envelop and keep-out geofences ensure separation from all manned and unmanned aircraft is maintained.

VII. Conclusion

This paper has discussed the application of two point-in-polygon algorithms, Ray Casting and Triangle Weight Characterization (TWC), to the problem of geofence boundary violation detection. Fast Ray Casting and TWC execute in approximately the same amount of time, but only TWC requires an initialization step. Results show that all three algorithms are consistently and rapidly able to accurately characterize UAS geofence violation status. Future work includes extension of TWC to evaluate boundary proximity and full consideration of dynamic geofence boundary updates.

References

[1] U. S. Government, “Operation and certification of small unmanned aircraft systems,” in *Title 14 Code of Federal Regulations, Part 107*.

[2] P. H. Kopardekar, “Safely enabling uas operations in low-altitude airspace,” 2017.

[3] M. N. Stevens and E. M. Atkins, “Multi-mode guidance for an independent multicopter geofencing system,” in *16th AIAA Aviation Technology, Integration, and Operations Conference*, 2016, p. 3150.

[4] A. Narkawicz and G. Hagen, “Algorithms for collision detection between a point and a moving polygon, with applications to aircraft weather avoidance,” in *16th AIAA Aviation Technology, Integration, and Operations Conference*, 2016, p. 3598.

[5] D. Alciatore and R. Miranda, “A winding number and point-in-polygon algorithm,” *Glaxo Virtual Anatomy Project Research Report, Department of Mechanical Engineering, Colorado State University*, 1995.

[6] K. Hormann and A. Agathos, “The point in polygon problem for arbitrary polygons,” *Computational Geometry*, vol. 20, no. 3, pp. 131–144, 2001.

[7] H. Rastgoftar and S. Jayasuriya, “Evolution of multi-agent systems as continua,” *Journal of Dynamic Systems, Measurement, and Control*, vol. 136, no. 4, p. 041014, 2014.

[8] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, “Triangulating a simple polygon,” *Information Processing Letters*, vol. 7, no. 4, pp. 175–179, 1978.

[9] D.-T. Lee and F. P. Preparata, “Location of a point in a planar subdivision and its applications,” *SIAM Journal on computing*, vol. 6, no. 3, pp. 594–606, 1977.

[10] MathWorks, “kml2struct - File Exchange - MATLAB Central,” accessed: 2017-02-25. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/35642-kml2struct

[11] ——, “deg2utm - File Exchange - MATLAB Central,” accessed: 2017-02-25. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/10915-deg2utm