Polygon Decomposition and Coverage Path Planning for Aerial Imagery

Amit Bendkhale[*], Ronak Chauhan[†], Varun Ramanathan[‡]

Abstract

Aerial imagery/photography has been established as a very useful technique for many activities such as mapping, surveying, creating orthomosaics, 3D-modelling, etc. Unmanned Aerial Vehicles (i.e UAVs or drones) are employed for this purpose . Our main objective was to investigate and implement optimal coverage path planning algorithms for aerial imagery using UAVs - that is, algorithms which return an efficient path for a drone so that the total time and the energy consumed during aerial imagery can be minimised. Convex polygons are the shapes for which it is easiest to find an optimal flight path, so we decided to partition the main polygon (representing the land to be imaged) into convex subpolygons and then give optimal flight paths for each of these smaller polygons. We provide implementation details for this algorithm and we finally discuss how this algorithm can be improved to consider further constraints on the flight path.

1 Introduction

The problem of area decomposition and coverage path planning for UAVs has been studied extensively. The main parts of the algorithm for CPP are

- Polygon decomposition
- Finding the optimal line sweep direction

Polygon decomposition is all about partitioning our polygon into subpolygons such that

- It is easier to find a flight path
- Each subpolygon can be covered in a single flight while satisfying other constraints

Since convex polygons can always be covered in an uninterrupted flight path (not taking into account the UAV's endurance), we minimally partition our polygon into convex polygons. For any convex polygon, we use the *lawnmower pattern* to cover the whole area. For fixed wing UAVs, the turns take more energy than the straight lines in the lawnmower pattern, so we try to minimise the number of turns. For minimising the number of turns, we find out the *width* of the polygon, which is the smallest vertex-edge distance for the polygon (this will essentially be the width of the smallest pipe in which you can fit your polygon). The straight lines of

[*]IIT Roorkee
[†]KJSIEIT, University of Mumbai
[‡]IIIT Delhi
The authors were interns at Chennai Mathematical Institute. The project was done in collaboration with Skylark Drones.

the lawnmower pattern are then kept parallel to this edge. Thus we ensure that the number of turns, which will be proportional to the polygon width, is minimised. Using the above algorithm, we can find the optimal flight paths for each of convex subpolygons. On paper, this seems straightforward to implement, but we must remember that these algorithms work on a 2D plane while the earth is far from 2D (it is a geoid). The use of *UTM coordinates* instead of the lat-long coordinates helps us solve this problem. The input for our algorithms were KML files, which contain the lat-long coordinates of the vertices. The final flight plans were to be prepared in "Mission Planner" which takes *poly files* as input, so we modified our algorithm to output poly files for each of the convex subpolygons. Our algorithm can be improved in many ways. We can sometimes recombine some convex polygons into a single concave polygon while maintaining an uninterrupted flight path. We can also take into consideration the endurance of our UAVs during polygon decomposition, and give optimal locations for *recharge points* – points where our drone can come for recharges.

Many uses of aerial imaging have *Ground Sampling Distance* requirements - the GSD is the distance between pixel centers when measured on the ground. Higher GSD values lead to less detailed images. Moreover, large variations in the height of the terrain can cause the GSD for the flight path to fluctuate out of the acceptable range. So, we can possibly modify our algorithm to ensure that in any single subpolygon, the GSD remains in the acceptable range.

The remainder of this paper is organised as follows: Section 2 explains the concepts and the algorithms for polygon decomposition as well as calculation of optimal line sweep direction; Section 3 throws some light on the implementation of the algorithm; Section 4 discusses various ways in which the algorithm can be improved.

2 Concepts

2.1 Image stitching & the lawnmower pattern

Photogrammetry involves obtaining a *photograph* of a particular area that is being studied and/or surveyed. These areas can be really large, and getting an image by clicking just one photograph from a very high altitude would actually increase the ground sampling distance (actual distance between centres of pixels), thus giving poor quality images. To overcome this, multiple photographs are taken from a lower height and are "stitched" together.

For this stitching to work, and to ensure that every point in the photograph has been captured from multiple perspectives (necessary for orthomosaics, 3D models, etc.), there must be some amount of *o*verlap between the images.

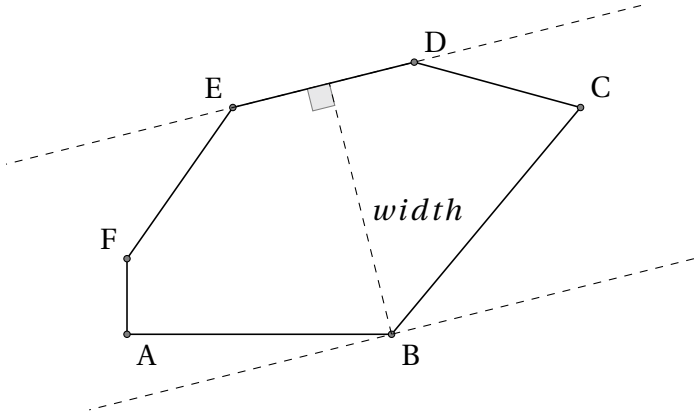
The flight path is basically a lawnmower pattern. The flight takes images in straight lines, or *stripes*. Each stripe ends with a U-turn that starts the adjacent stripe. Each stripe would overlap with both the adjacent stripes as per the percentage of overlap needed. This overlap between the stripes is called the *sidelap*. It is conceivable that the turns of a flight require more energy and time as compared to the rectilinear portions of the path since the UAV has to decelerate, turn and then again accelerate. It has been shown in [Li et al] that the turns do in fact consume more fuel. Moreover, no photographs can be clicked during turns. Thus it is in our best interests to minimize the number of turns while covering an area.

One such way is to consider the *width* of a convex polygon. The *span* of an edge E is defined as the greatest possible distance between E and a vertex V of the polygon. The *width* of a polygon is the minimum span of

any edge of the polygon.

$$\text{Width(P)} = \min [\ \text{span(e)}\ \forall\ \ e \in \text{Edges(P)}\ ]$$

However, it can also simply be defined as the least possible distance between two parallel lines such that the polygon remains in the interior (bounded) region between the lines.



We would like to implement the lawnmower pattern in our flight in such a way that we move parallel to the edge *E* and perpendicular to the width to minimize the total number of turns of UAV. We call this direction that is perpendicular to the lawnmower pattern's stripes as the *optimal sweep direction*. In the next section, we explain how we can compute the width of a convex polygon.

2.2 Width Calculation

2.2.1 Naive Algorithm

A naive approach for finding the width of any convex polygon is to find span for every edge by traversing all vertices of the polygon, and then get the minimum among all such spans. The time complexity of such an algorithm will be $O(n^2)$, where n is the number of vertices of our polygon. Pseudo-code for this algorithm:

Start ALGORITHM

CH = Convex-Hull (***P***)
for each edge ***E*** in ***CH***
    span = 0
    for all vertices ***V*** of ***P***
        distance(E,V) = 2*(Area of Triangle EV)/Length(E)
        if ***distance(E,V)*** > span :
            span = ***distance(E,V)*** and
            antipode = ***V***

if (span < ***width***) or (***E*** is first edge) :

width = span

---

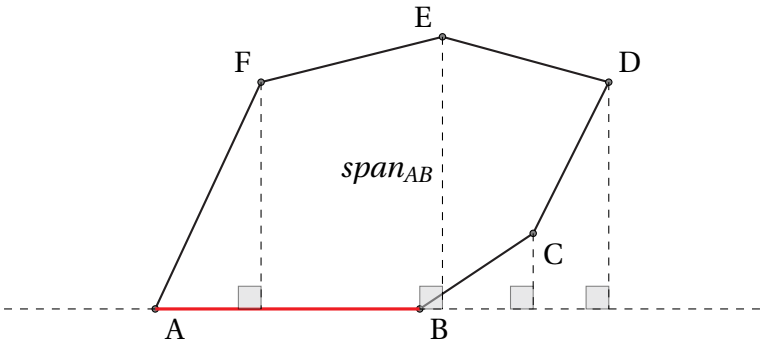
end ALGORITHM

2.2.2 Advanced Algorithm

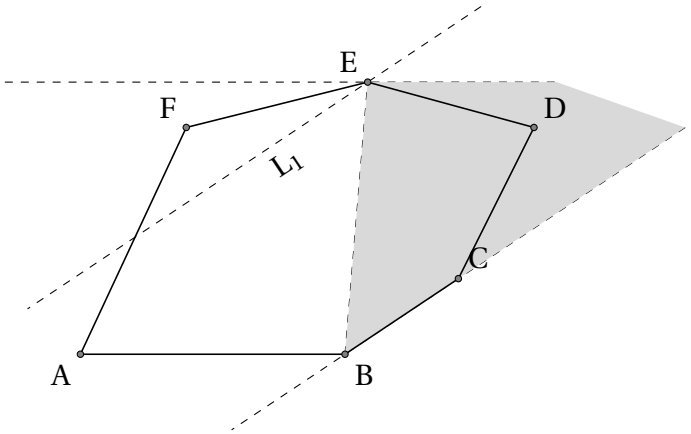
This is an amortized algorithm and is described in *Li. et al (2011)*. This algorithm removes redundant calculations from the naive algorithm for finding span of an edge and has an amortized time complexity of *O(n)*. Essentially, we rotate the polygon in counter-clockwise order to calculate span of every edge. For finding whether a vertex is an *antipode* (the farthest vertex) to a given edge, we check *distance-differences* of this vertex with respect to the previous vertex as well as the next vertex. If they have opposite signs, then the vertex is an antipode.

$$\text{Distance-difference } (V_1,V_2) = \text{dist (Edge E, } V_1) - \text{dist (Edge E, } V_2)$$

In the diagram shown below Distance-difference (C,D) and Distance-difference (D,E) are negative with respect to edge AB. But Distance-difference(E,F) is positive implying a sign change in distance-differences. Hence E is antipode of edge AB.



Now observe that if the antipode of edge AB is E, then the antipode of edge BC will never be B, C or D (i.e. vertices in the shaded region except the previous antipode - see diagram below). This is because E (the previous antipode) is further from BC than any point lying in bounded region of L_1 and extended BC. As B, C and D were explored earlier for finding span of edge AB, hence they won't be covered for edge BC. So only the remaining vertices (E, F and A) are traversed in counter-clockwise direction.



Start ALGORITHM

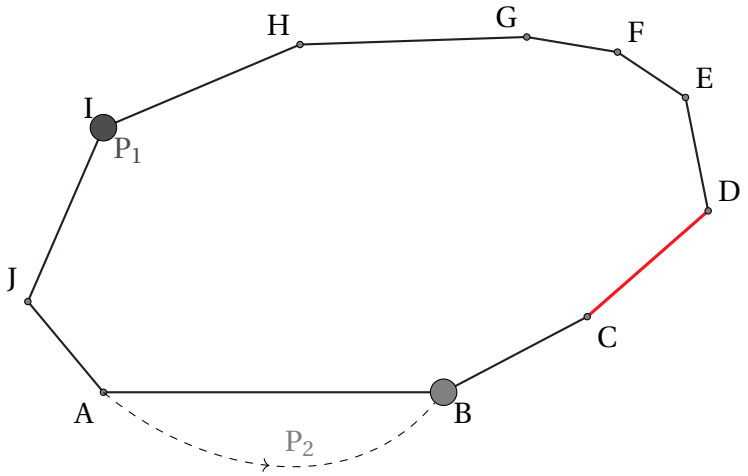
CH = Convex-Hull (***P***)
min-span = 0
n = number of vertices in ***CH***
antipode = array of zeros of size n
for *i* from 1...n :
    if (first edge) :
        j = prev-antipode = vertex[n]
    else :
        j = prev-antipode = antipode[prev-edge]
    found = 1
    while (found) :
        delta-prev = $(y_{i+1} - y_i)*(x_j - x_{j-1}) - (x_{i+1} - x_i)*(y_j - y_{j-1})$
        delta-next = $(y_{i+1} - y_i)*(x_{j+1} - x_j) - (x_{i+1} - x_i)*(y_{j+1} - y_j)$
        if ((*delta-prev* * *delta-next*) <= 0) :
            antipode[edge [i]] = vertex[j]
            span = distance(edge [i],vertex[j]) // = 2*Area of triangle (i,j)/edge-length(i)
            found = 0
        j = next CCW vertex
    if (span < ***width***) or (***E*** is first edge) :
        width = span

---

end ALGORITHM

Now let us try to understand why this algorithm runs in *O(n)* time. Consider a 10-sided convex polygon as shown below. We first assign two pointers P_1 and P_2 pointing at A. P_1 traverses the vertices (in CCW order) and stops when the next antipode is reached. P_2 also traverses the vertices in CCW direction, but it traverses

the next vertex only when the next antipodal vertex coincides with the previous one i.e. the two consecutive edges have the same antipode. We traverse edges in counter-clockwise direction starting at edge AB. Say P1, P2 are initially at A. We obtain the antipode of AB as G, hence P1 traverses 7 vertices (namely A, B, C, D, E, F, G) whereas P2 is still pointing at A. P1 again runs and finds that I is the antipode of edge BC, again P2 remains at A. Now P1 finds that the antipode of CD is I itself and hence P1 remains unchanged. But this time due to the repetition of an antipode (I here) P2 traverses to the next node, i.e. B. So P1 has traversed 9 vertices in total, whereas P2 has traversed 1 vertex. By the end of the algorithm, P1 would have visited each vertex at most twice since V has partitioned all vertices into two sets, namely those starting from A to prev(v), and those from next(v) to J. Hence, V can be an antipode of edges in either sets. Moreover, P2 visits atmost n vertices because the total number of times that any antipode gets repeated must be less than or equal to the number of edges, which is n. Hence, the total number of vertices visited by both P1 and P2 is atmost 3n which is O(n). Since calculation at each vertex takes a constant amount of time, the amortized time complexity of the algorithm is O(n).



3 Methods for polygon decomposition

In most concave polygons, an uninterrupted flight path is not possible. To overcome this problem, we decompose such a concave polygon into a set of convex subpolygons. We calculate the width of all these subpolygons, get optimal sweep direct their optimal line sweep directions, and then use our UAV to photograph each of these subpolygons separately.

We discuss 2 approaches for decomposing concave polygons into convex subpolygons. The key idea is to partition the polygon at concave points, thus removing the concavity.

- Naive Decomposition
- Enhanced Exact Cellular Decomposition

Note that both these approaches are *methods* rather than fixed algorithms, and hence their complexities would depend on their implementations.

3.1 Naive Decomposition

The key idea here is to extend both edges of concave points to intersect some side of the polygon. We calculate the sum of widths of the resulting subpolygon for both the cases and make the cut which results in a smaller sum.Pseudo-code for the algorithm:

Start ALGORITHM

---

for each point ***P*** in *Polygon.Points*
    if ***P*** is a concave point
        extend one of the sides and find intersection ***I1*** with some side of the polygon
        find sum of widths of resulting polygons : ***w1***
        extend the other side and find intersection ***I2*** with some side of the polygon
        find sum of widths of resulting polygons : ***w2***
        if ***w1*** < ***w2*** :
            make cut along the line segment joining ***P***, ***I1***
        else :
            make cut along the line segment joining ***P***, ***I2***

---

end ALGORITHM

3.2 Enhanced Exact Cellular Decomposition

For every concave vertex, we do the following. Starting from this vertex, we extend a line segment parallel to an edge E_p of the polygon, and we do this till our line intersects the polygon. Then, we calculate the sum of the widths of the resulting subpolygons. We do this for all edges instead of just E_p (that is, we try extending a line parallel to every edge) and then use the decomposition which results in the minimum sum of widths. For more details, one can refer to Li et al's paper ([3]) on EECD. Pseudocode:

Input: *Polygon P*
Output: *Set of convex subpolygons C*

Start ALGORITHM

C = empty_set()
for each cell ***C_i*** in ***C***:
    if ***C_i*** *has concave vertices*:
        minimal_width = MAX_VALUE
        vertex_index = 0
        line_index = 0
    for concave vertices ***i*** :
        for border_lines ***j*** :

erupt a support line from ***i*** parallel to ***j***
connect ***i*** with the closest reachable intersection
(SP1, SP2) = the new subpolygons
width(i, j) = ***width(SP1)*** + ***width(SP2)***
if ***width(i, j)*** < minimal_width:
    minimal_width = ***width(i, j)***
    vertex_index = ***i***
    line_index = ***j***
cut the polygon in ***vertex_index*** with line parallel to ***line_index***
call these two subpolygons ***c1*** *and* ***c2***
C.add(***c1***)
C.add(***c2***)
C.remove(***C_i***)

---

end ALGORITHM

4 Implementation

We first studied the concepts of width, optimal line sweep direction, along with naive and EECD algorithms for polygons. Based on our study, we intended to implement the EECD algorithm in Python. For this a <Polygon> data-type was implemented, along with algorithms for finding the convex hull and width of the polygon. Later, based on our interaction with employees working at Skylark Drones, we learnt the following.

1. The input is a KML file (which is basically an XML file with longitude-latitude co-ordinates and other information that is read by Google Earth).
2. This KML file is overlayed on the map in an application called Mission Planner. Based on this overlay, a polygon can be stored in a plain text file with lat-long co-ordinates. Mission Planner can read, write, and perform any calculation using polygon files.
3. The polygon can be very large; we were given a 517 sided polygon which had an area of about 13000 acres, hence the actual running time of the algorithm matters a lot practically.

This is when we realized that our implementation of the EECD algorithm would be very inefficient and slow for real world applications due to Python being an interpreted language. Hence we used a polygon partitioning algorithm from the open source library CGAL (Computational Geometry Algorithms Library), which is a C++ library. According to CGAL's documentation, the algorithm is credited to [6].

We are given lat-long co-ordinates, which are different from Euclidean x-y co-ordinates. Round-offs in lat-long co-ordinates can lead to drastic deviations in terms of the distance to be travelled and hence the area to be swept by the UAV. Here we learnt about the UTM (Universal Transverse Mercator) co-ordinate system, which essentially maps the lat-long co-ordinates from the earth's ellipsoid to a cylinder.

We used the utm package in Python to convert lat-long co-ordinates to UTM, ran CGAL's algorithm to get partitions, and converted UTM co-ordinates to lat-long. Additionally, we noticed that Mission planner generates a flight path in the optimal line sweep direction. Hence we decided to use the path generated by Mission Planner.
Key implementation steps can be described as follows:

1. Read kml file with long-lat co-ordinates
2. Convert these to UTM co-ordinates
3. Run the polygon partitioning algorithm from CGAL
4. Output a polygon file with lat-long co-ordinates for each partition
5. Use Mission Planner to plan flights for each partition, vary altitude, etc

5 Future work

5.1 Considering flight endurance

So far we have considered that the UAV will be able to cover the entire area of a convex subpolygon in one flight. However this may not be the case when the area is too large. In such conditions the endurance of the UAV must be taken into account. Based on this, a convex polygon may need to be partitioned further so that the resulting subpolygons can be covered using uninterrupted flights.

5.2 Recombination

For a given polygon, if we have extremely small partitions, the number of flights needed increases a lot. One way to counter this is to *recombine* partitions that have same line sweep direction. Again, we would need to consider the endurance of the UAV.

5.3 Variance in terrain height

This work deals with 2D polygons representing agricultural farms, where the terrain is mostly a plain stretch of land. But in cases of hilly regions or for other applications where there is a lot of variation in terrain height, the GSD (ground sampling distance) might vary outside the desired range when the UAV flies at a constant altitude. To resolve this, we could partition the polygon to ensure that no subpolygon has a height variation outside the desired range. Applications like Google Earth have an extrapolated estimate of the height at any point on the earth's surface, so once could possibly extract this information and use it while decomposing the polygon.

6 Bibliography

[1] Jan Bulušek's - Coverage Path Planning in Non-Convex Polygon Areas for Orthophotomap Creation Using UAVs, *Diploma Thesis at Czech Technical University in Prague, 2017*

[2] Marina Torres, David A. Pelta, José L. Verdegay, Juan C. Torres - Coverage path planning with unmanned aerial vehicles for 3D terrain reconstruction, *Expert Systems with Applications, February 2016*

[3] Yan Li, Hai Chen, Meng Joo Er, Xinmin Wang, Coverage path planning for UAVs based on enhanced exact cellular decomposition method, *Mechatronics*, Volume 21, Issue 5, 2011, Pages 876-885

[4] Marina Torres, David A. Pelta, JoséL. Verdegay, Juan C. Torres, Coverage path planning with unmanned aerial vehicles for 3D terrain reconstruction, *Expert Systems With Applications* 55 (2016) 441–451

[5] L. H. Nam, L. Huang, X. J. Li, J. F. Xu. An Approach to Coverage Path Planning for UAVs, *2016 IEEE 14th International Workshop on Advanced Motion Control (AMC), DOI: 10.1109/AMC.2016.7496385*

[6] Daniel H. Greene. The decomposition of polygons into convex parts. In Franco P. Preparata, editor, Computational Geometry, volume 1 of *Adv. Comput. Res.*, pages 235–259. JAI Press, Greenwich, Conn., 1983.