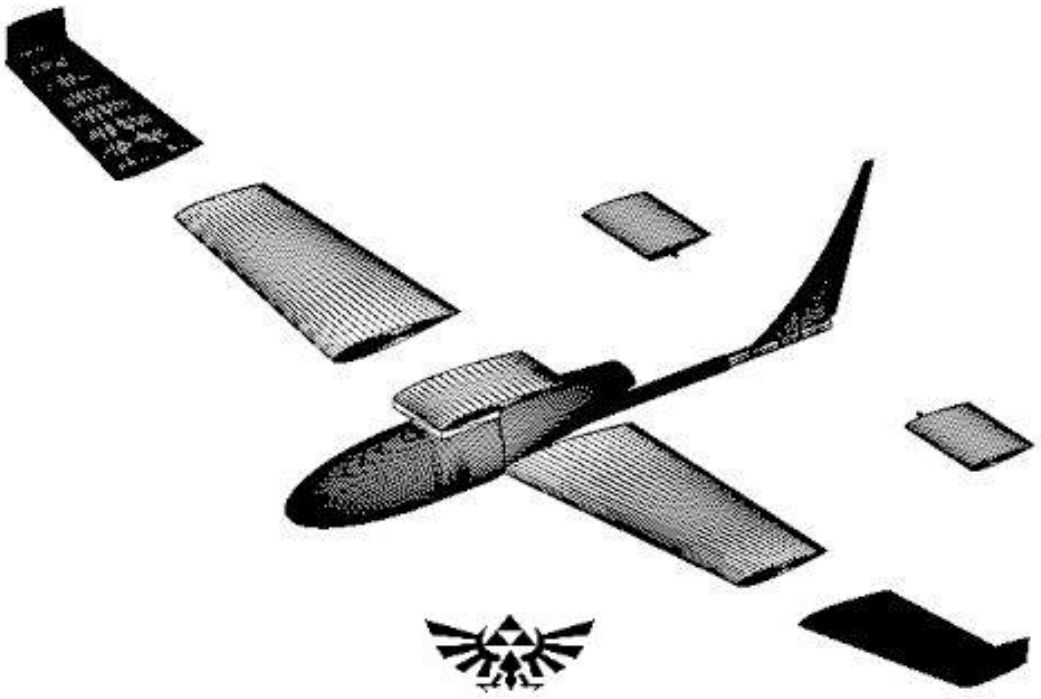
Computer-based Modelling 2: A Parametric UAV

Jerome Wynne Alex Tross Youle Emma Milner
Jordan Bellamy Ciro Garcia-Agullo Jocelyn Roberts
Lucy Sayes Val Ismaili Johan Crook Alex Maddison

Term: December 2015. Report date: 1st December 2015.

Abstract

This report details the process and outcome of a two-month project to cooperatively develop a Matlab programme that generates the a 3D-printable unmanned aerial vehicle (UAV). The overall approach to solving the problem is first outlined, then the code used to execute the solution is presented and explained for each component. Screen-shots of the final virtual models and images of the printed components are included, as is an evaluation of the project's successes and shortcomings.
There is overlap in the approaches used by each author, and it should be noted that there was extensive cross-fertilisation of ideas within the team.



Contents

1 Introduction

This section aims to define the rationale behind the group's design formulation and explain the code such that anyone unfamiliar with the program could easily glean a working understanding of it.
The team's first decision, which had the largest influence on the entire project, was to split the UAV into components that could be connected together, so that the entire assembly could be built using a variety of printer sizes.
Research was conducted to look into typical UAV shapes and designs to aid the development of our conceptual design generation. We decided on a traditional model similar to that of the Raven UAV, which is approximately the size of a remote-control plane. We aimed to make certain parameters bespoke so the user would not be limited to our design and could change the UAV's dimensions according to the task required of it.
All team members began by familiarising themselves with controlling three-dimensional plots in Matlab, using the NACA aerofoil generator available on Mathworks' File Exchange, and reviewing a similar project that 3D-printed a global gear. Research into the most flexible means of generating the UAV's geometry was conducted, and two possible methods identified.
The first of these methods was straightforward - generate vertex data, plot a surface, then use surf2stl (available on the File Exchange) to write an stl file from that data. This was a simple method, that had proven effective, however it did not appear to offer the same degree of control over the form created as the alternative method.
The alternative method involved generating the vertex data, but then manually specifying the faces connecting these vertices. Unlike the previous method, this approach required manual triangulation between points, ensuring that the face normals of these triangle pointed in the correct direction. This method was more laborious and required a more innovative approach, however it paid dividends in the clarity of the resulting plot, and the precise control it offered over the bodies produced. Furthermore, it was clear exactly how this method was working, whereas surf2stl seemed to behave more as a box of black magic.
The decision on method divided the team, but was ultimately inconsequential - both methods worked, and neither presented any major problem to the team. In hindsight, dividing up the components was a mis-step - communication between pairs was poor, and it was often found that parts did not fit together properly, resulting in the need to hard-code unpleasant-looking scale factors into each function. Instead, a more group-focused approach where the assembly was considered paramount from the first instant, would have been more appropriate.

2 Operation Instructions

The following instructions are to be used as a guide for the intended user as a walk-through from the initial stage of UAV customisation down to final stage of attaining the .stl files needed for rapid prototyping. They will allow the user to tinker with various configurations and immediately view the changes made through both exploded and assembled viewpoints.

1. Open all included files.
2. Identify the file 'GUIFinal.m' and run.
3. Having now launched the Graphical User Interface, all customisation of the UAV and generation of .stl files will be done through interaction with this newly opened window.
4. Initially, there is no displayed assembly but only a set of controls. If the user has no desire to customise the design, predetermined values have been included in the configuration panel allowing the user to simply click 'Generate' in the actions panel and see the default design.
5. Clicking the 'Explode/Assembly' button will allow the user to see the separation between sections. Alternatively, the tabs visible on the left hand side of the screen allow for close up looks at the key sections of the UAV.
6. In the circumstance that the user wishes to customise the design, they have the ability to do so by inputting values in the configuration panel.
7. Changes in 'Wing span' value will not only effect the wings but will automatically cause the fuselage and connection piece to scale appropriately.
8. Wing profile is customisable by inputting NACA four-digit values into 'Main Wing NACA'. The 'Dihedral' angle of the wings can also be controlled.
9. 'Taper' input defines the tapering of the main wings, with a maximum value of 0.265.
10. 'Tail height' and 'Elevator span' are available as separate inputs as they are not automatically scaled by changes in wing span.
11. Any changes can be erased through the use of the 'reset all' button.
12. Once satisfied with the existing parametric design, .stl files can be created using the 'Write .stl files' button.
13. Finally, transfer files and 3D print.

3 The Main Wings

3.1 Wing - Exterior

When Engineers try to analytically break down aerofoils, the preferred method is to plot points around the upper and lower surface and 'dot-to-dot' plates between them; allowing aerodynamic and fluid hand calculations and further working. A function called naca4gen has been used here to generate upper and lower surface data co-ordinates for given 4-digit NACA designations in a similar fashion. This function is an integral part of the code for the front wing sections - providing consistently accurate geometries for every aerofoil that a manufacturer might desire. The data aligns the two-dimensional aerofoil in the x and z axes, which forces manipulation in a third dimension to be made in the y axis.
So as to model this three-dimensional solid, the first lines of code were aimed at projecting an image of the aerofoil along the y. This idea was further developed with a focus on integrating a taper feature into the design of the wing. Preliminary input values were created to define necessary dimensions:

```
length = 800; %Span of one wing
terminus_width = 200; %Chord width at root
levels = 50;
taper = 0.15; %Decimal gradient of taper
\label{lst:WingExtValues}
```

These values were modified upon amalgamation into the GUI, so as to reflect upon a relationship with a designated wing span length and the interior wing sections. For purposes of this report, these preliminary values will be used.
In the case whereby one image is projected and a surface was surfed in between, the only possibility to incorporate tapering would be in the form of a scale-factor resizing the projected co-ordinates, giving a linear shrinkage or augmentation. This is the easiest solution to the problem, but research into ideal wing shapes showed that the most efficient wing types are elliptically designed[1] - providing the best ratio between the lifting area and mass of material used in manufacture. It was decided that the best method to implement this feature would be to divide the length into a series of increments, using the levels value to determine the number of gaps. By using a linearly spacing (linspace) and a matrix replicating (repmat) command, as shown below, we were able to equidistantly place image steps along the wing span in the y axis and then copy the two variables that map aerofoil geometry, x and z, to each level.

```
% Create y-coordinates for each level
y = linspace(0,length,levels); %Spacer
y = repmat(y, size(x,1), 1); %Replicates the y values down the
    columns
%until y matches the column length of the z and x matrices
% Replicate z and x coordinates for each level
x = repmat(x, 1, levels);%Replicates x along each row to match y
z = repmat(z, 1, levels);%Similarly matches z to the levels in y
```

By sub-dividing into increments, it is possible to apply scaling factors that aren't necessarily linear, such as exponential or quadratic; making for interesting results. Starting with something more gentle, the general formula for a linear taper is as follows:

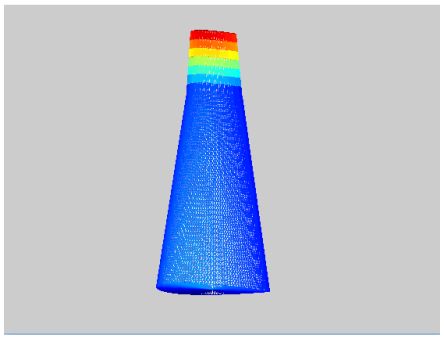
```
phi = terminus_width - taper*y;
```

```
x = x.*phi; %Scales each x-value to the taper dimensions
z = z.*phi; %Scales each z-value to the taper dimensions
```

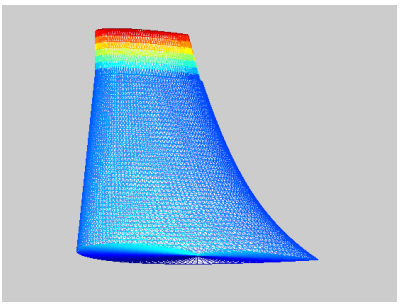
Phi is an array designed to change the x and z co-ordinates of each level by taking the original root width - terminus_width - and subtracting a function of the wing length multiplied by a decimal modifier. The value for taper was set previously in the code as 0.15 for this example. If it was desired to have a more creative wing taper, the code could be modified to become:

```
g = y/y(end); %A new arbitrary variable
phi = terminus_width - terminus_width*(tan(g-4*pi/3));
```

g is created by making each level in y a decimal fraction of the total length - the final input of y. By experimentation, subtracting a combination of the tangent function multiplied by the root width from the root width produced some interesting results, shown in Figure [1]. With further experimentation, it would be possible to come up with any number of possible combinations; to the point whereby a user could input a desired equation into the GUI and formulate a shape. Regrettably there was not enough time to implement this feature; the time to manually write out individual equations was found to be too taxing on a limited project with as great a learning curve as this.



(a) This figure shows a linear taper where the scale of the projected image is a function of the wing span



(b) There is huge potential for exponential or trigonometric functions to define the taper in the GUI such as the tangent function creating this taper

Figure 1: Two very different tapers

The midpoints of both the first and final faces that have been found with the function face_centre.m (explained in Section 4.3) are then appended to the co-ordinate arrays:

```
% Add tip and base mid-point coordinates to our matrices
xm = face_centre(x, [1 levels], np); % A function to figure out
    the mid point of aerofoils automatically
x = [x(:); xm(1); xm(2)]; % Stick the x mid-points coordinates
    onto the end of our existing x matrix
y = [y(:); 0; length;]; % Stick the y mid-points coordinates
    onto the end of our existing x matrix - these are the base
    and tip y values
z = [z(:); 0; 0;];
cp = size(x,1); % Get the size of our complete matrix of points
```

At this stage, once the increments have been described and the appropriate z and x co-ordinates mapped, the process of triangulation is needed. By mapping a series of triangles

between points on a plane, it is possible to mesh a surface that can be consequently translated from MatLab to a .stl file. Using triangulation provides a simple method of surfacing objects within minimal lines of iteration, and has proved to be a more foolproof method of ensuring that MatLab recognises a solid object by working via an easier co-ordinate basis than surfing.

To describe the surface over the wing, the following code was used:

```
%% Create the triangles describing the wing's faces
T = [];
% Create the faces around the wing's edges
for c = 0:levels-2
    for k=1:np-1
        t = [k k+1 k+np; k+1 k+np+1 k+np];
        T = [T; t+(c*np)];
    end
end
```

The iteration works between the first and final face, and 'stitches' together each face with corresponding triangles. This process nicely wraps triangles around the entire objects by filling an array of T as an wx3 matrix of three points that would need to be meshed together. In a succinct fashion, by filling this array T with points from all functions, the assembly can collect together data for every mesh in the system and perform a trimesh operations in one place.

This iteration is then repeated for the two aerofoil images at base and tip of the wing. Triangles are created from the midpoint and two adjacent points on the upper and lower surfaces until the whole image is covered.

```
% Create the faces at the tip and base of the wing
for k=1:np-1
    T = [T; k k+1 cp-1; cp-2-k cp-1-k cp];
end
T = [T; 1 np cp-1; cp-1-np cp-2 cp];
```

Once these triangulation iterations are complete, a closed volume can be modelled and appropriate .stl files created.

3.2 Wing - Winglets

There are many elements of aircraft design that cause aerodynamic drag. A particularly troublesome component of drag on the wings is induced drag, caused by vortices at the tip of the wings. These vortices cause a swirling flow in the fluid that is stronger at the tips and weaker towards the start of the wing (at the fuselage). Induced drag is due to the distribution of lift along the aerofoil length. The less this distribution changes, the less the induced drag on the wing. A vortex at the tip causes a greater change in lift along the wing because of the difference in flow and so will cause more induced drag.

Winglets can be used to reduce induced drag because they weaken the vortices by smoothing out the swirling flow of the fluid at the tip of the wing and increasing lift. Winglets also cause the aspect ratio of the wing to increase. The aspect ratio of a wing is the ratio of wingspan to chord. The wingspan is the length from one wingtip to the other and the chord is the thickness of the wing (or the length from the farthest point on the leading edge of the aerofoil to the tip of the trailing edge). Increasing the aspect ratio is

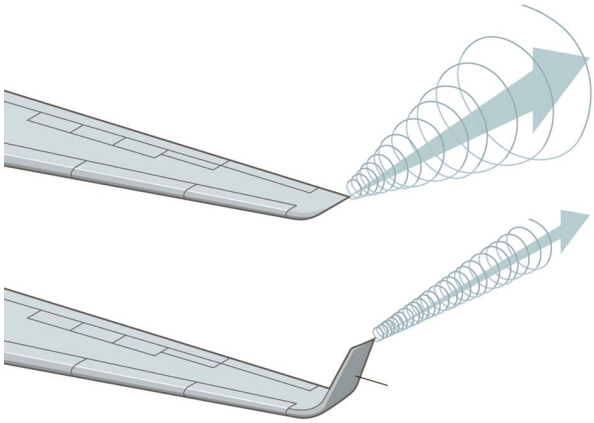


Figure 2: As shown, winglets work to reduce the vortex spiralling off the wing, increasing the efficiency of the wing.[2]

usually done by making the wing longer and thinner but there comes a point where this reduces lift by too much and becomes impractical. A winglet can be added instead to have the same effect without having to change the wingspan. Induced drag is inversely proportional to the aspect ratio and therefore by adding a winglet and increasing the aspect ratio, you cause the induced drag to decrease. This is why it was decided that a winglet should be included as an option in the design.

[3][4]The angle of the winglet is optimum when the winglet is vertical with respect to the ground. This is because lateral lift is reduced. However if the wing itself is on an angle that is not perpendicular to the fuselage then the angle of the winglet must be adjusted so that it is still vertical. Therefore 90deg is not the only angle the winglet can be. For this reason the angle of the winglet automatically changes in the GUI with the angle of the wing while still optimising the lift-drag ratio with their winglet.

```
1 winglet_length = length*0.15;
2 winglet_alpha = 90; %winglet angle
3 if winglet_alpha == 90;
4     winglet_alpha = 89.9999;
5 end
6 winglet_ss = find(y >= length - winglet_length);
```

So far an aerofoil of a certain length has been made. A number of faces will be chosen at the end of this aerofoil to turn through an angle to create the winglet. The find function is used to define all the points which are going to form our winglet. The winglet length is a percentage of the length of the whole wing (line 1). The if function is used to catch errors if the angle is 90 then it will change it to 89.9999. This is done because in line 2 of the following code, the trigonometry will not work with an angle of 90deg (as cos(90) = 0).

```
1 b1 = y((iaf.n.*2)+2) - y((iaf.n.*2)+1); %Pinpoints step size
2 b2 = b1*cosd(winglet_alpha);
3 b = b1 - abs(b2); %difference needed - abs() avoids negative
    values
```

One problem that quickly arose was that the winglet length would increase to far more than we needed, particularly for larger angles. See Figure [**??**]. This came about because the code specified that no matter the true length of the wing, the span in the y direction had to be maintained. The angle would therefore tilt the whole wing up such that there was a projected image that was the length MatLab desired.

In order to fix this problem, a solution was devised that would reduce the incremental step size in the y direction such that when the z co-ordinates were projected up at at the desired angle, the new hypotenuse would be the same length as the previous increment size - which looks far more natural and maintains the dimensions of the wing to the best quality that could be coded in 4 or 5 lines.

b1 is a slightly convoluted method of finding the incremental step size. The formula for the number of points in an image happened to be the sum of the upper and lower points on the aerofoil (twice iaf.n) plus one repeated data point. By subtracting the y values of the first and last points of adjacent images, it was possible to determine how long each step was.

To find b2, it was assumed that b1 was the hypotenuse of a triangle, which is the desired solution of the problem. By using trigonometry with the winglet angle, b2 was found as the required length of each new step.

This function is the reason that the if function earlier changes an angle of 90 to 89.9999 because $\cos 90° = 0$ and therefore this would not produce a useful transformation.

Finally, b is the difference between b1 and b2 so it can be incrementally removed from every y step past the winglet point. To do so, a new array of values, , was needed to act as a multiplier of b, such that for every new step past the winglet value, another multiple of b was subtracted to correct the displacement. f was found by taking every value of y of the winglet, subtracting the last value that was not part of the wing, and then dividing by the step size.

```
1  f=(y(winglet_ss)-y(winglet_ss(1) - 1))/b1; %Multiplier array
2  y(winglet_ss) = y(winglet_ss) - f*b; %Multiplier*difference
       to correct lengths
3  z(winglet_ss) = z(winglet_ss) + tand(winglet_alpha).*(y(
       winglet_ss) - (length-winglet_length-b));
4  z(winglet_ss) = z(winglet_ss) - z(min(winglet_ss));
```

In order to project z upwards or downwards via the trigonometric method planned, it was necessary to bring it a lot closer to the origin such that the length of y could be multiplied by the tangent function without creating a ridiculously large and incorrect value. Therefore, each z value was projected upward in this way. As a failsafe, should there be issue with the z values not being centred around the origin, it would be re-centred to the origin by subtracting the smallest value of z, which always happens to be the midpoint of the first aerofoil which was appended to the end of the array further up in the code.

3.3 Wing - Interior

The interior section of the wing is composed of two components: a complete aerofoil section, midwing11, and a half section divided along the chord of the aerofoil, midwing2. These are illustrated in figure 4. During the concept generation phase, it was decided that the most effective way of connecting the wing interior and the wing-fuselage connector was to use a male female connection, hence the half section which allows the connector to integrate into the wing. This relationship is displayed in figure 6.

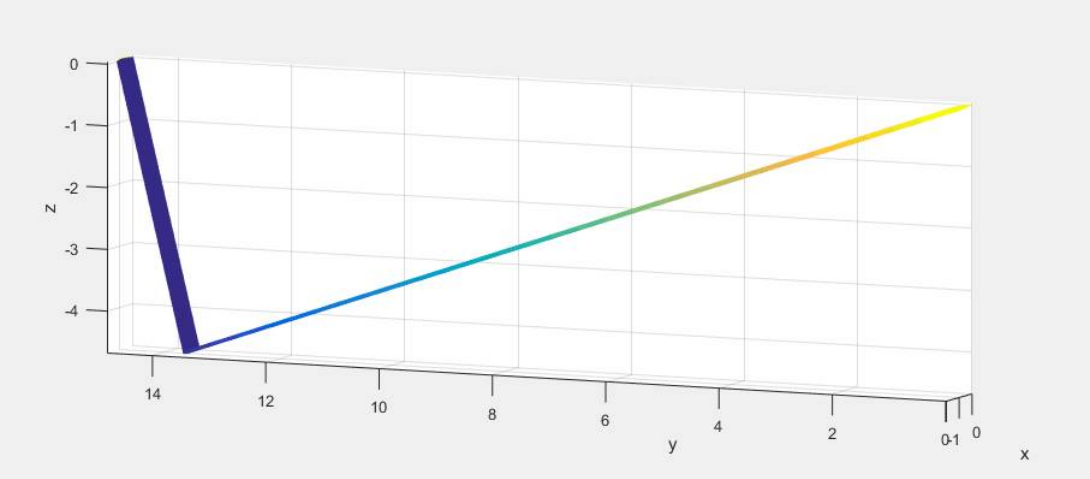


Figure 3: Here the winglet angle is set to 40deg but the length has also dramatically increased

The method used to generate a 3D body of an aerofoil used the NACA Aerofoil Generator

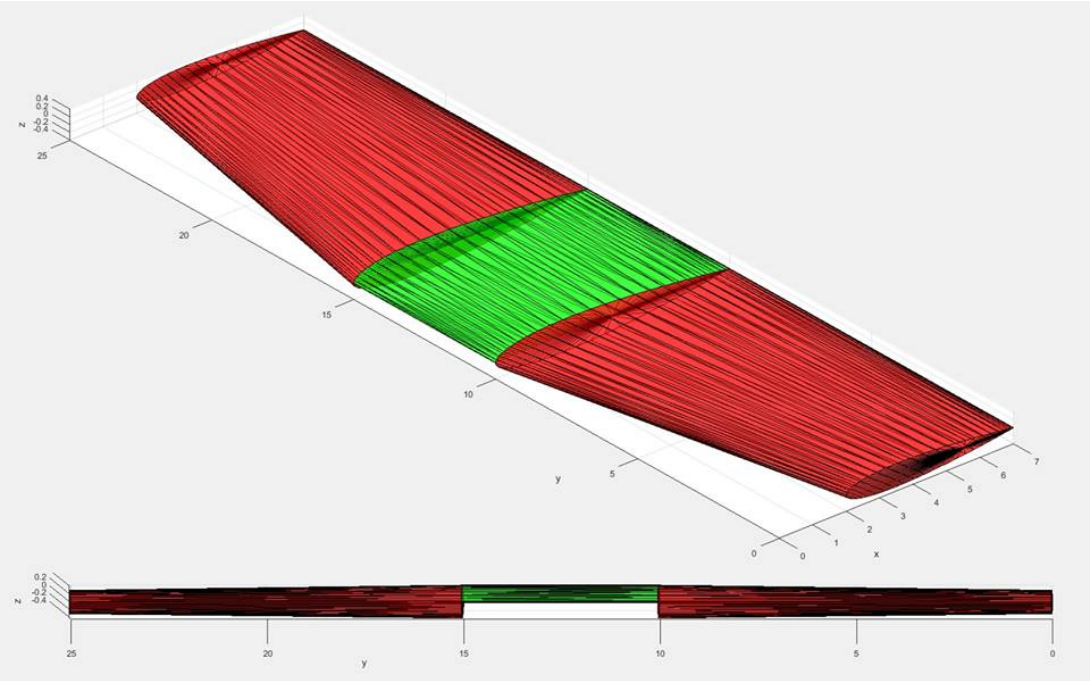


Figure 4: The assembled midwing section.

to create a two-dimensional shape. The aerofoil array was replicated and displaced a distance L from the original 2D aerofoil.

```
x = [af.xU; af.xL]; %combines upper and lower x elements
%of the aerofoil array
z = [af.zU; af.zL]; %combines upper and lower z elements
%of the aerofoil array
y = zeros(size(x)); %generates y array size of x of zeros
y = [y; 0; L*ones(size(x)); L]; %defines y array on face
%y = 0 then adds another y array the size of x at
```

```
%distance L from 1st face (defines second cross-section
%of face)
x(:,2) = x; %generates second x array
x = W*x; %scales x by W
z(:,2) = z;
z = W*z;
```

As shown in Section 3.1, the wings are coded such that an optional taper is available. The slope, or gradient, of the taper is defined by the user. For continuity, the taper of the inner and outer wing segments must be equal, as to avoid vertices or sharp changes in direction of the wing. This was achieved by scaling the inner face by T.

```
x(:,2) = x(:,2)*(W - taper*L)./W; % Generates taper of x and z.
    determined by wing span
m1 = max(x(:,2));
m2 = max(x(:,1));
x(:,2) = x(:,2) + m2 - m1; %translation of array in x axis to
    account for scaling differences.
z(:,2) = z(:,2)*(W - taper*L)./W;
```

However, after scaling one of the faces, the wing tapered along the trailing edge. This presented a problem, as the purpose behind tapering this section of the wing was to provide a smooth, continous edge with a taper of constant angle over the entire wing span. By tapering the leading edge, a bend was present at the connection between mid-wing and outer-wing. To rectify this, a translation of one of the faces was required. This was done by calculating the difference between the maximum x element of both scaled and unscaled faces using the max() function, generating the red components of Figure 4.
In order to mesh the two cross-section faces of the aerofoil, the midpoint of the aerofoil had to be determined in terms of x and z and added to the end of their respective arrays.

```
end_upper_z = size(af.zU, 1); %defines final value of z
%array in NACA generator.
end_upper_x = size(af.xU, 1); %defines final value of x
%array in NACA generator.

midpoint_z = (z(1)-z(end_upper_z))/2; %define midpoint of
%1st and last z values
midpoint_x = (x(1)-x(end_upper_x))/2; %define midpoint of
%1st and last x values
x = [x; midpoint_x;]; %adds midpoint to end of x array
z = [z; midpoint_z;]; %adds midpoint to end of z array
```

Subsequently, a column vector was developed using an iterative process to generate a number of points which were later used to connect the two aerofoil shapes using triangulation. A complication presented by the traingulation method was the order the vertices of the defined triangles. Initially the order was incorrect, resulting in the surface facing into the body, such that the generated stl file could not be printed. This was tested in the Makerbot software. The vertices were initially specified as [k k+1 k+np; k+1 k +np k+np+1], adjusting it to [k k+np k+1; k+1 k+np+1 k+np] reversed the direction the surfaces faced.

```
T = []; %creates empty array
```

```
2
3  for k = 1:np-2
4      T = [T; k k+np k+1; k+1 k+np+1 k+np]; %creates points
5      %in T array for triangulation function for aerofoil
6      %surface
7  end
```

Following this another iteration took place which generated the coordinates used for the trimesh (triangular mesh) between the individual coordinates of the aerofoil and the midpoint of said aerofoil.

```
1  t=[]; %creates empty array
2
3  for k = 1:np-1
4      t = [t;k k+1 np]; %creates points in T array for
5      %triangulation function for aerofoil cross-section
6      %face
7  end
```

To translate these coordinates such that they apply to the displaced aerofoil section, the t array was replicated. The new elements were increased in magnitude by a value of np, which is defined as np = size(x, 1), i.e. t = [t; t+np].
The vectors generated in the two iterations were then combined to allow the use of the triangulation function in Matlab. The triangulation function requires column vectors of equal sizes to run. However as shown above the x and z variables are Nx2 matrices. It was therefore necessary to convert them to column vectors:

```
1  x = [x(:,1); x(:,2)]; %converst x and z arrays
2  z = [z(:,1); z(:,2)]; %in column vectors
3  T = [T; t]; %combines T and t arrays so i can use single
4  %variable in triangulation funtion
5  TR = triangulation(T, x, z, y); %generates triangles
6  %between specified points -->vertices created = stl
7  %conversion possible
8  tm = trimesh (TR); %generates mesh of triangles
```

Triangulation generates triangles between predefined points, which consequently establishes vertices and a collection of faces on the model. This is especially important when exporting the code as an .stl file, as the defined vertices and faces allow the body to be 3D printed.
As the inner wings are used on both sides of the UAV, it was neccessary to reflect it. When assembling the UAV, the orientation of the innerwing can be changed according to the side of the UAV. As shown below, if the component is to be postitioned on the right-hand side, the aerofoil will adjust appropriately.

```
1  if strcmp('right', side)
2      y = -y;
3  end
```

Producing the half section aerofoil used the same process as stated above, however the component is not tapered or reflected and points extracted from the NACA function differed slightly:

```
x = [af.xU; af.xC]; %af.xC, af.zC extract array for
z = [af.zU; af.zC]; %aerofoil chord from af. structure
```

Elements for the upper half of the 2D aerofoil and the x and z elements of the aerofoil chord array were extracted from the NACA af. structure.
Overall the midwing component performs well. It is flexible and works adequately for a large variety of aerofoil shapes and sizes. It can be defined be 3 variables: aerofoil type (e.g. 0012), aerofoil depth (W) and length(L), reducing any complications regarding the full assembly, as length and depth were defined with respect to the overall wingspan. Some possible improvements of the code could allow for parabolic tapering used in the outer wing described in Section 3.1 and potentially the option for a dihedral at the connection between midwing11 and midwing2.
Incorporating a dihedral was decided upon at the early stages of the project. The initial approach in doing so was to generate an angle joint which would connect the inner and outer wings. The joint, shown in Figure 5, used the same concept as stated above - two displaced aerofoil cross-sections connected via triangulation. The dihedral was executed using a rotation formula on the x and z arrays.

```
%rotate second aerofoil plane
y(:,2) = y(:,2)*cos(p) - z(:,2)*sin(p);
z(:,2) = y(:,2)*sin(p) + z(:,2)*cos(p);
```

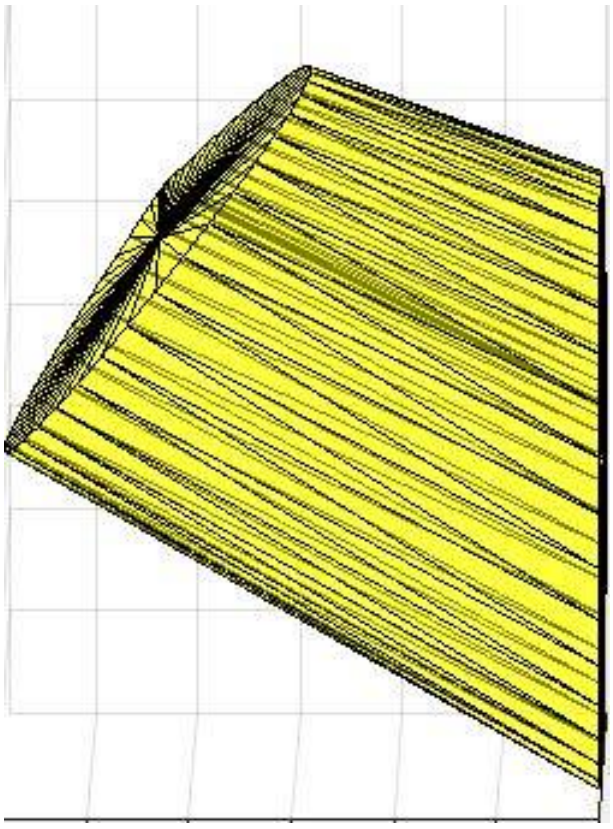


Figure 5: The initial approach to developing a dihedral.

However, it was established that the dihedral could be generated using the code for the outer wings described in Section 3.1. The joint was obsolete and therefore discarded. The mechanism used to create the dihedral increased the z coordinate of each layer generated in the outer wing (Section 3.1), while maintaining the overall length of the wingspan.

```
z=z+(tand(Dimensions.dihedral).*y.*cosd(Dimensions.dihedral));
```

In the above line of code, z is the height of the points in the wingtip and tand is the tan of the dihedral angle in degrees. Maintaining the cross-section area of the aerofoil in the initial rotation joint was an issue and was occasionally unreliable. The component also struggled with aerofoils wit significant cambers. By translating the aerofoil layers in the z axis, no rotation was necessary, removing the area complication and making the code very flexible.

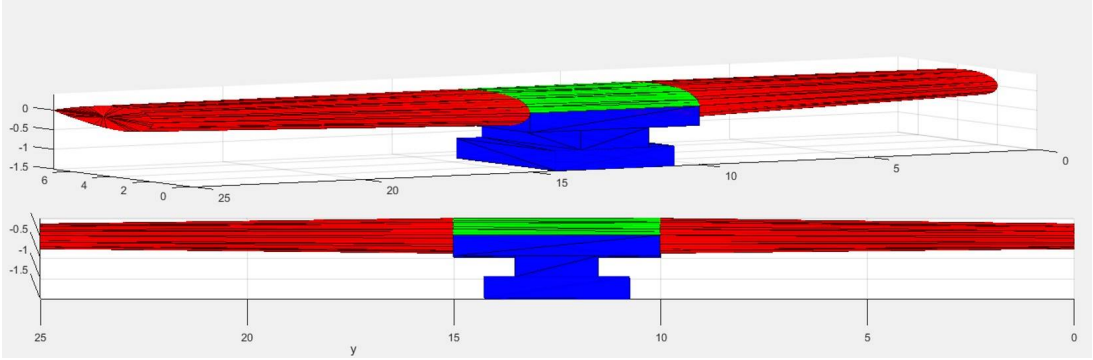


Figure 6: Illustrates the interaction between the fuselage-wing connection and the mid-wing section.

3.4 Wing - Fuselage Connection

The connection piece between the wings and fuselage is a small but important feature of the design of the UAV. It allows for a more elegant solution to the connection of the two pieces than simply gluing them together and was therefore developed as a preferable solution (see figure 7). The chosen design was a male-female connection, with the piece being discussed in this section being the male connector with the female counterpart being included at the top of the fuselage. This meant that communication between the two groups was critical to ensure the consistency of both shape and dimensions in order for the two pieces to precisely fit. Therefore, the dimensions of width (w) and depth (d) of this piece were dictated by, and equal to, the width and depth of the half sectioned aerofoil detailed in the previous section. Consequently, this allowed the dimensions of the female connection to be defined - further detailed in section 4.1. An alternative approach to the

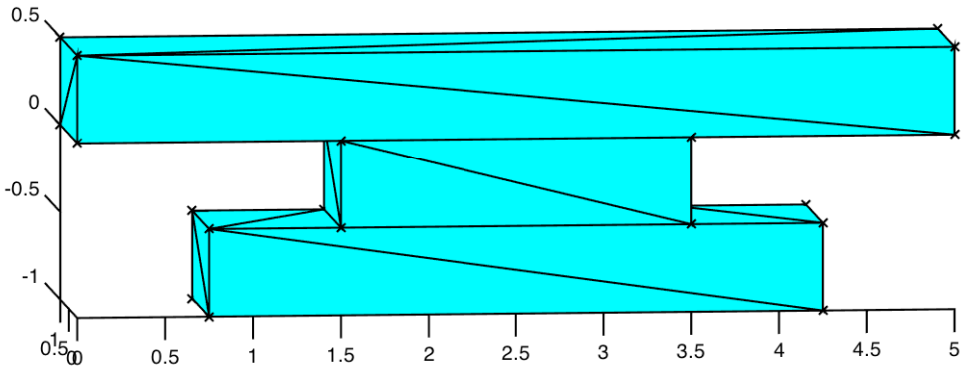


Figure 7: Male connector for the connection of wings and fuselage

the automated style method of triangulation (discussed in previous sections) was used. The reason for this was due to the simplicity of the shape which meant a manual method of triangulation was a reasonable process.

To begin with, the x,y and z coordinates of every vertex of the desired shape were defined and ratioed as shown below.

```
x = [-2 6 6 -2 -2 6 6 -2 1 1 3 3 1 1 3 3 0 0 4 4 4 4 0 0]*w/4;
y = [0 0 1 1 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]*d;
z = [0 0 0 0 2 2 2 2 0 0 0 0 -5.65 -5.65 -5.65 -5.65
     -5.65 -5.65 -5.65 -5.65 -6.85 -6.85 -6.85 -6.85]*h/3;
```

At this stage, the shape was simply a collection of points and was lacking the faces to provide an actual outline of the connector that was possible to both plot and print. Therefore, the method of triangulation was used to create the necessary faces of the shape. This was done through the creation of a matrix containing the indices of every 3 points to be connected for all 36 triangles, as shown by the following code:

```
T = [1 2 5; 1 4 5; 4 5 8; 2 5 6; 5 6 7; 5 7 8; 2 3 6; 3 6 7;
     3 4 7; 4 7 8; 1 4 9; 2 11 12; 4 9 10; 2 3 12; 9 10 13;
     10 13 14; 11 12 15; 12 15 16; 14 17 18; 13 14 17; 16 19 20;
     15 16 19; 19 20 21; 20 21 22; 18 23 24; 17 18 23; 22 23 24;
     21 22 23; 9 13 15; 9 11 15; 17 21 23; 17 19 21; 18 20 22;
     18 22 24; 10 12 16; 10 14 16];
```

When called upon in the final assembly the piece was able to be placed in the correct position with the dimensions accurately ratioed resulting in the connection visualised by the graphical user interface.

4 The Fuselage and Tail

4.1 Fuselage

The original design for the fuselage was an amalgamation of a hemisphere, cylinder and cone. As this was a very simple fuselage, iterations of the design occured throughout the project, with the final fuselage being far more complex and aerodynamic.

To create the nose, a hemisphere was used;

```
x1 = x1(1:round(end/2), :);% Creating a hemisphere
y1 = y1(1:round(end/2), :);
z1 = z1(1:round(end/2), :);
```

This was then slightly compressed along its y-axis using the line of code y1 = 0.8∗ cylinder_radius∗y1;, and then elongated along the z axis; z1 = 2∗cylinder_radius∗z1;. Each coordinate in the matrix of the hemisphere was multiplied by 'cylinder radius', which was a variable defined by the user originally, but when the part was integrated with the other parts of the UAV, this became defined as a result of a scale factor and the wing length.

In order to give this hemisphere a thickness, to enable 3d printing, an inner layer with a slightly smaller radius was created; r2 = cylinder_radius−(cylinder_radius∗0.1). The main body of the fuselage had both an inner and an outer layer to match those of the nose, and was created using the cylinder function.

It was essential to create a way of joining the fuselage to the wings. A design was decided on between the fuselage and wing design teams, in which a T-shaped connecting

part on the wing would slide into the fuselage. This was initially planned to be exterior to the fuselage, and this part was created as a separate part using triangulation. However in a later iteration of the design it was decided that this was too bulky and would cause too much drag, so the part was changed to lie inside the fuselage, enabling the wing to sit smoothly on top of the body, minimising drag. The shape of the connection slot is shown below (Figure 8).

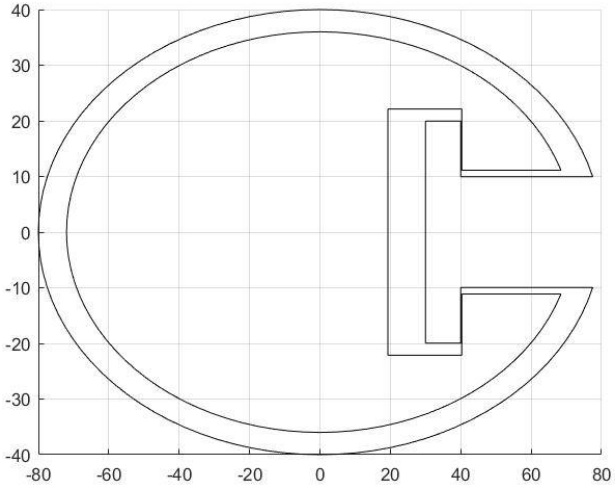


Figure 8: An example of the two layers of the T shaped connection.

The connection slot had to be created at the top of the fusealge cyclinder, about point 1 of the cylinder matrix. Thus the points either side of point 1 (in both the inner and outer layers) needed to be manipulated in order to create the desired slot shape. Due to this being the first point of the cylinder, each half of the hole had to be created separately, as one half used the last 5 points in the matrix and the other used the first 5, as seen with the code here which was used for the inner layer.

```
for i=1 : round(5) % Iterative loop selecting the first four
    points on one side of the central point of the cylinder.
x3(:, i) = (4*scale_factor)+(0.1*3); % Moves points 1 to 5 down
    slightly lower than the outer layer (4*scale_factor)
% shall be, to enable a thickness to be created.
end

% This section of code creates one half of the required T shape
    of the hole by dragging each point
% to a certain position by use of a factor of the scale_factor
    in order to match the attachment piece.

y3(:, round(5)) = y3(:, round(6));
y3(:, round(4)) = y3(:, round(5))+(scale_factor+(0.1*
    scale_factor));
y3(:, round(3)) = y3(:, round(4));
x3(:, round(3)) = x3(:, round(4))-(2*scale_factor+(0.1*
    scale_factor));
```

```
x3(:, round(2)) = x3(:, round(3));
x3(:, round(1)) = x3(:, round(2));

for i= 1 : 2 % This loop ensures that the central point is level
    with point 2.
    x3(:, i) = x3(:, round(2));
end

for i= round(N-3) : (N+1); % Iterative loop selecting the first
   four points on the other side
% of the central point of the cylinder.
x3(:, i) = (4*scale_factor)+(0.1*3); % Moves the points to the
   same level as was set for points 1 to 5.
end

% This section of code creates the other half of the required T
   shape of
% the hole by use of the same method as before.

y3(:, round((length(y3))-4)) = y3(:, round((length(y3))-5));
y3(:, round((length(y3))-3)) = y3(:, round((length(y3))-4))-(
   scale_factor+(0.1*scale_factor));
y3(:, round((length(y3))-2)) = y3(:, round((length(y3))-3));
x3(:, round((length(x3))-2)) = x3(:, round((length(x3))-3))-(2*
   scale_factor+(0.1*scale_factor));
x3(:, round((length(x3))-1)) = x3(:, round((length(x3))-2));

for i= round(N) : N+1 % Selects the last 2 points of the
   cylinder.
    x3(:, i) = x3(:, round((length(x3))-1)); % Makes the points
       level with the point before.
end
```

The same was then done for the exterior cylinder but only using the 4 points either side of point 1, with slightly smaller displacements in order to preserve the thickness for 3d printing .

Prior to targeting the specific points, a percentage of the total points the cylinder was made from were used to select which points were used. However, when the resolution value was less than 70 the connection shape did not join up, and when greater than 130, the shape of the cylinder started to distort. Although the final code does not have resolution as a variable input, at this stage it was intended to be. Prior to the scale_factor being introduced, the manipulated x values were dragged to a certain value, which would have been an additional limiting factor as the cylinder_radius could not have been set at this value or smaller.

The initial connection slot concept was also meant to extend through the cylinder body into the motor housing. This was changed because, although it was feasible, there was an issue in scaling due to the motor housing cylinders radius getting smaller. This resulted in a change of approach in which the fuselage was split into two sections; the tail and motor housing, fuselage_tail , and the body and nose, fuselage_body. These parts would then

be glued together after the wing had been attached to the body, creating a seal that would prevent the wing from being able to slide out.
Originally, the fuselage tail was a simple cone shape with a hole to accomodate the tail of the UAV, however in a later design iteration it was decided to increase the complexity to accomodate a motor. this resulted in a design with two offset cones, which had slighlty eliptical bases in order to fit smoothly to the fuselage body. In order for the propellor to be able to rotate freely without hitting the UAV tail, the two cones were offset away from each other; the tail connector to the bottom of the fuselage and the motor housing to the top, using the method below; for the fuselage tail

```
x6(3, :) = x6(3, :) - (0.45*cylinder_radius);
```

and for the motor housing

```
x7(2, :) = x7(2, :) + (0.3*cylinder_radius);
```

As with the fuselage_body, both cylinders in the fuselage_tail have an inner and outer layer to enable 3D printing, shown in the following image of the tail connection hole (Figure 9).

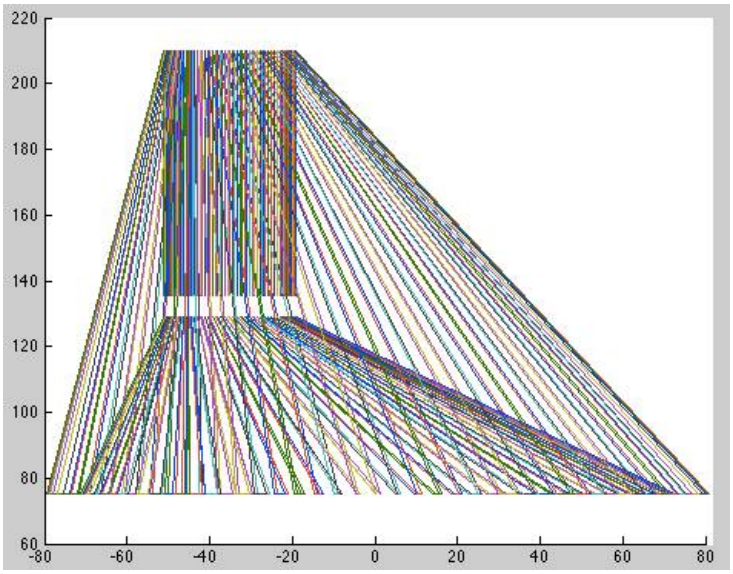


Figure 9: An image showing the inner and outer layer, as well as the connection hole

.

As you can see, the inner layer does not have a hole as there is no need, however it is not a solid block as we decided to make the piece as hollow as possible to reduce the weight of the part.

Both the tail hole and the motor housing scale with the UAV size, although the way in which they do this is different. For the tail, the radius of the hole scales in relation to the cylinder radius, and defines the radius of the tail connector piece. The depth of the hole is defined by the the tail length, as shown below.

```
z6(5, :) = z6(1, :) + 0.45*tail_length; % Set depth of tail
    connector hole
```

The motor housing scales by the user inputing the motor dimensions into the GUI, which enables different motor sizes for different sizes of UAV. For example, the motor length is defined in the GUI, and the line of code below creates the hole depth.

```
z8(4, :) = z8(2, :)- motor_length; % Set depth of motor hole
```

Although there was an attempt to use triangulation within the fuselage section, it was decide that due to the use of two layers throughout, it did not make sense to use this technique, as it frequently caused issues. The function surf2stl worked perfectly for the way the part was created, despite it not being consistent with the technique used in the rest of the parts. If given more time, the wing attachment slot would have preferably extended into the tail as this would have allowed the fuselage to have been just one piece.

4.2 Tail Connector

The tail connector joins and supports multiple parts of the UAV. It connects to the fuselage, tail fin and the rear wings. To allow for the differences in how each part is designed and to simplify the assembly process the tail connector was designed to be adaptable. To be consistent with most of the other functions and parts within the UAV, triangulation was used to create the surfaces around each shape. The initial function that was developed is called prismAM.m, this function allowed the user to produce different prisms. It first defined all the points within the prism, for example a cylinder, before each of the points were joined by triangulation.

```
%Cylinder%
    [x,y,z]=cylinder(radius,100);
    n=length(x);
    x=[x(1,:) x(2,:)];
    y=[y(1,:) y(2,:)];
    z=len_cylinder*[z(1,:) z(2,:)];
```

The way the triangulation process was conducted allowed the shape of the prism to be changed very easily; the only difference between different prisms are the co-ordinates and number of vertices. To demonstrate this and allow the function to be adaptable an if statement was inserted.

```
if is equal(type,'c')
```

The function allowed the user to select one of three shapes; a cylinder, an octagonal or a hexagonal prism, although more shapes could have been used. The type of prism generated was specified by the first input into the function (c, o or h).
The points representing the prism had already been defined in a format which the triangulation function could read so the next part of the triangulation process was to define the faces. Using the number of points (n) around one edge of the end shape the process of defining each triangle was automated.
This worked by creating a matrix (faces), using the zeros function, and populating it by defining each column of the matrix in turn. Each element defines one vertex of each triangle, with each row defining one complete triangle and the element's value identifies which point the function should connect. The increment function (:) was used to create a pattern which allowed adjacent triangles to be created very quickly. Similar functions define the triangles around the side and the ends of the prism. However, to bridge the gap between the start and the end of the triangles around the sides of the prism it was required to manually input the values of some vertices (lines 20-27 below) as these triangles use the first and last points defined by the shape functions and so do not follow the pattern which was used to specify the other triangles. All of this is shown below.

```
%Create faces matrix%
faces=zeros((4*(n-2))+2,3);
%FACES: Cover end%
faces(1:n-2,1)=1;
faces(1:n-2,2)=(2:n-1);
faces(1:n-2,3)=(3:n);
%FACES: Cover other end%
faces((n-1):(2*(n-2)),1)=n+1;
faces((n-1):(2*(n-2)),2)=(n+2:n+(n-1));
faces((n-1):(2*(n-2)),3)=(n+3:2*n);
%FACES: Cover sides%
faces((2*(n-2))+1:(3*(n-2)),1)=(2:n-1);
faces((2*(n-2))+1:(3*(n-2)),2)=(3:n);
faces((2*(n-2))+1:(3*(n-2)),3)=(n+2:n+(n-1));

faces((3*(n-2))+1:(4*(n-2)),1)=(3:n);
faces((3*(n-2))+1:(4*(n-2)),2)=(n+2:n+(n-1));
faces((3*(n-2))+1:(4*(n-2)),3)=(n+3:2*n);
%FACES: Cover sides - bridge gap%
faces((4*(n-2))+1,1)=2;
faces((4*(n-2))+1,2)=n+1;
faces((4*(n-2))+1,3)=n+2;

faces((4*(n-2))+2,1)=1;
faces((4*(n-2))+2,2)=2;
faces((4*(n-2))+2,3)=n+1;
```

The triangulation and trimesh functions were then used to create and display the final prism. The result of this function when a cylinder with the radius of 1 and length of 1 is specified can be seen in Figure 10. This system was copied and used throughout the development of the tail connector, with each surface of every shape being triangulated in a very similar way.

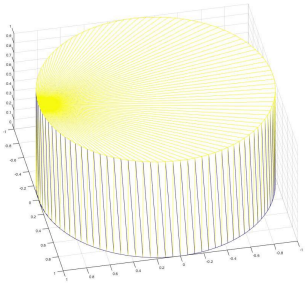


Figure 10: Cylinder created using prism function

The next step taken was to make the prism hollow. This involved defining a smaller similar prism within the existing prism. The variable thick was added to the function to allow the wall thickness to be defined by the user. To reduce the complexity of each functions' inputs for each different type of prism a new function was created (e.g. cylinderAMhollow.m and octagonAMhollow.m). Like before, each wall of the shape was

defined in the FACES matrix as a series of triangles with the sides of both the inner and outer prisms being defined in the same way. However, the ends of the prism are slightly different as the whole of the side is not being covered so it is not possible to have all the triangles fanning out from a single point like before. The result of the cylindrical hollow prism can be seen in Figure 11.

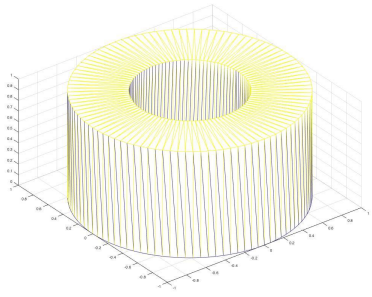


Figure 11: Hollow cylinder created using cylinderAMhollow.m

To aid in the assembly and functionality of the final UAV it was decided that to join the different components onto the tail connector a peg and hole system would be used. This allows the rear wing to rotate around a point and it helps ensure that the tail fin is securely fitted in the correct position. The holes would be situated within the tail connector. As the tail fin and the rear wing would be situated around the same distance from the start of the tail connector it was logical that the rear part of the connector would be an octagonal prism, so that the holes could be perpendicular to each other without spanning any edges.
To create a hole a new small cylinder was created using the cylinder function and it was positioned so that its ends lay flush with the sides of the hollow octagon into which the hole is being punched. This happens after the faces which the hole is being punched through have been deleted. The points defined by this function were then added onto the end of the existing sets of points and the FACES matrix was extended to allow for the triangles need to create the hole.

```
%Delete faces for hole%
  faces((8*(n-2))+1,:)=1;
  faces((8*(n-2))+2,:)=1;
  faces((8*(n-2))+3,:)=1;
  faces((8*(n-2))+4,:)=1;
%Create and position cylinder%
  [x_cy,z_cy,y_cy]=cylinder(hole_rad,100);
 m=length(x_cy);
  x_cy=stx+[x_cy(1,:) x_cy(2,:)]+0.5*oct_len;
  y_cy=sty+thick*[y_cy(1,:) y_cy(2,:)];
  z_cy=stz+[z_cy(1,:) z_cy(2,:)]+0.5*height_oct;
%Add cylinder onto end of existing points%
  x=[x x_cy];
  y=[y y_cy];
  z=[z z_cy];
faces=[faces; ones(2*m+2*(m-2)+32,3)];
```

A for loop was then set up which ran through every point within the newly created cylinder. Within each loop a vector was created of the distances from the point (within the cylinder) being considered and each point within the octagonal prism. The min function was then used to find the lowest value within that vector and its position.

```
[close_val(i),close_pos(i)]=min(20+(((x_cy(i)-[x1  x1]).^2)+
((z_cy(i)-z(1:2*n)).^2)).^0.5);
```

This position corresponds to the position of the point (on the octagon) within the x, y and z vectors which is closest to the point on the cylinder being tested; so in the FACES matrix the point on the cylinder can be connected to the point on the octagonal prism which is closest to it as well as the point next to it.

```
faces(8*(n-2)+i+8,1)=close_pos(i);
faces(8*(n-2)+i+8,2)=4*n+i-1;
faces(8*(n-2)+i+8,3)=4*n+i;
```

The majority of the face around the hole is now covered by triangles, however using just the code detailed above leaves 4 large gaps around the hole (Figure 12).

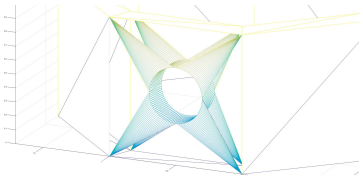


Figure 12: Octagonal prism with incomplete face around hole

To overcome this an if statement was used to identify the four points closest to the edges of the octagonal prism and to connect them to vertices either side of the edge. This process was then repeated again for the side of the hole inside of the hollow prism, before the face within the hole itself was created. A complete octagonal prism with a hole inserted into one of its sides can be seen in Figure 13.

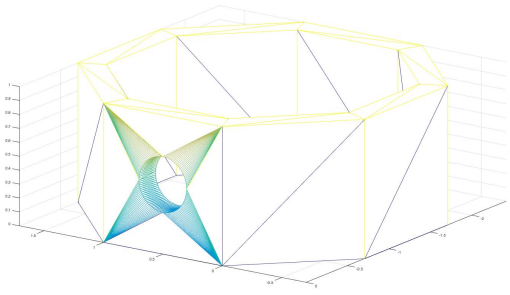


Figure 13: Octagonal prism with hole

Another version of this function was also created with two holes inserted into opposite sides of the octagonal prism to allow for the rear wing pegs to run all the way through the tail connector.
Another decision that affected the structure of the tail connector was to 3D print the UAV, to accommodate the length of the tail connector and allow it to be 3D printed it was split into 2 parts. The front section of the connector was to be a cylinder and the rear an octagon with the aim of sizing both prisms to create an interference fit between them.

The final assembly of the tail connector is very modular and uses multiple functions to allow for different sizes of cylinder and octagonal prisms, as well as the holes for the tail fin and rear wing being situated in different positions. The functions that are used are cylinderAMhollow which creates a hollow cylinder; octogonAMhollow which creates a hollow octagonal prism; octogonAMhollow_hole_top which creates a hollow octagonal prism with a single hole for the tail fin and octogonAMhollow_hole_bothsides which creates two holes for the rear wing. The use of these modular functions allows the tail connector to be very adatable, as was required, proucing an effective component of the UAV and increased its usefulness. However this adaptability increases the complexity of the program and it can be difficult to troubleshoot and use. An example of the final tail connector can be seen in Figure 14

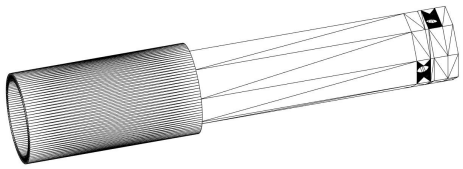


Figure 14: Example of final tail connector with holes for tail fin and rear wings

4.3 Tail Fin

The tail fin was developed over several iterations, and the development process included numerous proof-of-concept functions. The final result can be seen in figure 15. The fin has an exponentially tapering profile along a path with a linear gradient, and has two discrete sections: the front, where the male connector for connecting to the tail connector is, and the flap, which is a yaw control surface.

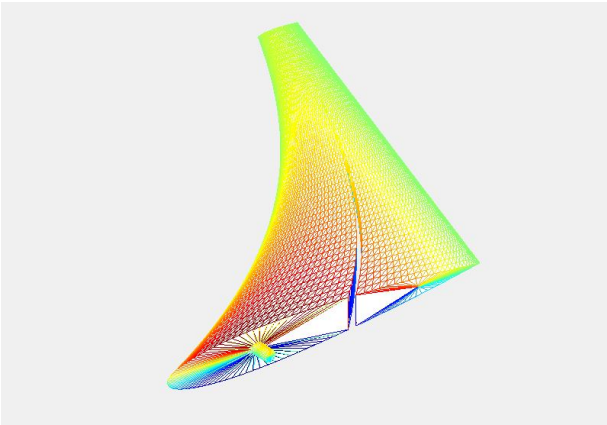


Figure 15: Final tail fin piece. Male connector on front, rear section is control flap.

The first tail fin model developed, generated by the function tailfin_with_holes .m, was blocky and would have been inadequate for use in an actual UAV on account of its poor lift characteristics. However, this function showed that the triangulation method had the

capacity to allow nuanced control over the part, and would produce sharply resolved plots of the final model. Furthermore, it also acted as a testing-bed for the first triangulation algorithms.
As can be seen from figure 16, the tailfin model produced by tailfin_with_holes .m has two holes passing through it. These holes were not hard-coded - instead, the generic function holierThanThou.m incorporated them into the existing shape's geometry.

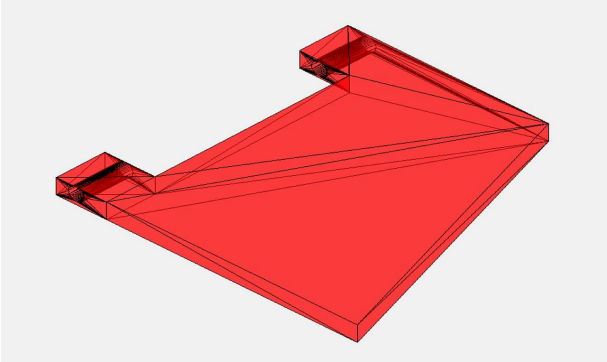


Figure 16: Early sketch of the tail connector, utilising a generic hole function.

holierThanThou.m took seven inputs: two 4x1 vectors containing the indices of the points describing the faces that the hole is created between, two ordinates stating where the hole's center should be relative to the first point of the first face, the radius of the hole, and its resolution (i.e. the number of points describing its circumference). Using these, the function performs the following sequence of operations:

1. Remove the existing faces bounded by the vertices in the input vectors. RemoveTriang ulations .m is used for this; it is written such that the 2x3 matrix of triangulations `Tr` handed to it as argument is removed from the main matrix of triangulations `T`, i.e.

```
function Tu = Remove_triangulation(T, Tr)
    for n = 1:size(T,1)-1
        if T([n n+1], :) == Tr
            T([n n+1], :) = [];
            sprintf('Removal succesful');
            break
        end
    end
    Tu = T;
end
```

   Where `Tu` is the updated triangulation matrix that does not contain the faces specified by `Tr`.

2. Create a new face with a hole in the position given by the arguments that have been handed to it. This requires two major steps - performing the triangulations between the faces bounding the hole and the hole itself, and creating the triangulations between the two circles describing either end of the hole. The algorithms for this were clunky in comparison to those used later on in the project such as in wibblyHole .m and do not merit discussion here.

One of the major shortcomings of generic_hole.m was that it could only create circular holes between quadrangular surfaces. A natural development of this function then, was to give it the capacity to generate holes on oddly-shaped surfaces, and the function Better_holes does that. This function works by first generating polar coordinate pairs, then applying some function that describes the desired geometry to these pairs. For example, figure 17a demonstrates the result of describing the border's radius using the function

```
xr = 20 + thb.*abs(cosh(thb))/100;
```

where `xr` is the set of radial values describing the outer circumference of the object.
Other examples of betterHole.m's capabilities are shown in figures 17b and 17c.
The triangulation method used for the irregular end faces was a crucial step towards triangulating between the tail fin's aerofoil boundary and the root of its connector. The problem was to find a means of triangulating between the hole and any bounding geometry; a bounding geometry such as an aerofoil. The approach used in the final tail fin function and the betterHoles.m function just discussed was the following:

1. Centre the hole and the bounding geometry at the origin on a set of polar axes.
2. Compare the θ ordinate of a point describing the hole with all points in the boundary, saving the difference .
3. Get the subscript of the boundary point with a θ ordinate that is closest to the given hole point.
4. Create two triangles between the selected hole and boundary point and the next point in the sequence at both the hole and the boundary.
5. Repeat steps 2-4 for each point in the hole.

This algorithm is equally applicable to a cylindrical connector (such as that seen in figure 18) as to a hole, and appeared in Tail_fin as

```
max_camber_ss = find(af.z == max(af.z));
[aero_bound_th, aero_bound_r] = cart2pol(xf - af.x(max_camber_ss
    ), zf);

p1_last = [];
for k = 1:size(aero_bound_th,1)-1
    difference = abs((aero_bound_th(k)) - th_con);
    p1 = find(difference == min(difference));
    T = [T; k k+1 p1(1)+cp];
     if k~=1
         if p1 ~= p1_last(1)
             T = [T; p1(1)+cp p1_last+cp k];
         end
     end
    p1_last = p1(1);
end
T = [T; 1 cp+p1(1) points_fore; cp+p1(1)+1 cp+p1(1) 1];
```

Where max_camber_ss was the subscript of the point where the fin's camber was largest. The reason for centering the connector about this was simple - it offered the greatest space and therefore allowed the connector to be as large as feasible, making for the strongest connector possible. An alternative connector arrangement was considered where there were multiple connectors of smaller radii, however the algorithms required quickly became unwieldy, and it became apparent that using multiple connectors was infeasible for the tail connector.

(a) Generic hole function example piece, gathered from betterHoles.m.

(b) betterHoles.m result with a high-frequency sinusoidal function.

(c) betterHoles.m result with a low-frequency sinusoidal function.

After refining various triangulation techniques using further test functions similar to betterHoles.m and holierThanThou.m, the final version of the tail function was created. The method for triangulating between the connector and the base has been discussed, and the other triangulation operations performed were similar to those seen elsewhere in this report. The top and bottom faces were triangulated between all points around their boundary and a point placed in their centre. The geometry of the tail fin's bulk is described using a series of scaled and truncated aerofoils. The scaling allows a taper to be produced, whereas the truncation separates the fin into a front section and a flap. The span-wise faces, i.e. those connecting the series of scaled aerofoils, were triangulated using the following algorithm:

1. Connect point to the point above in the cross-section directly above, and to the next point in the current cross-section.

2. Connect the next point in the current cross-section to the point directly above that, and to the point directly above the current point.

3. Repeat 1-2 for each point in cross-section.

4. Repeat 1-3 for all cross sections, bar the last.

In Tail_fin.m, this algorithm is codified as

```
T = []; % Triangulation matrix
for c = 0:levels-2
    % Faces for the front part
    for k=1:points_fore-1
```

```
5           t = [k+tp k k+1; k+1 k+tp+1 k+tp]; % Two triangles
                between this and the next level
6               if k == points_fore-1 % If at the last point on the
                    fin's front section
7                   % Create the triangle between the first and last
                        point describing the front
8                   t = [t; k+2+points_rear k+1 1; k+1+tp
                        points_fore k+2+points_rear];
9               end
10          T = [T; t+(c*tp)]; % Make those triangles connect level
                c to level c+1
11      end
12
13      % Faces for the flap
14      for k = points_fore+1:tp-1
15          t = [k k+1 k+tp; k+tp k+1 k+tp+1];
16          T = [T; t+(tp.*c)];
17      end
18  end
```

The function's performance is consistent across all NACA values, and is positioned consistently within the assembly, as should be expected. Precluded features that would be desirable in future versions are:

- Holed supports either side of the flap, in a similar configuration to the rudimentary tail fin in figure 16.
- A connector that wraps around the tail connector rather than slots into it. This would take advantage of the octagon-circle interference fit we designated to use.
- A triangulation algorithm that can create faces between curved surfaces. This is an unusual problem that would require an innovative triangulation algorithm. A possible solution might be

    1. Plot a circle that is in the x-y plane.
    2. Plot the curved surface beneath the circle.
    3. Use the function describing the curved surface to calculate the z-values at each point describing the circle, and translate the circle to these points.
    4. Find the point outside of the circle that is closest in theta and radially to a given point on the circle.
    5. Use Matlab's built-in delaunay triangulation capabilities to triangulate the surface up to the boundary described by these closest points.
    6. Triangulate between the circle points and the nearest points.

4.4 Rear Wings

The modelling of the tail wing of a UAV in Matlab can be viewed as an extrusion of a cross-section of an airfoil. The approach followed to model the tail wing was to mimic

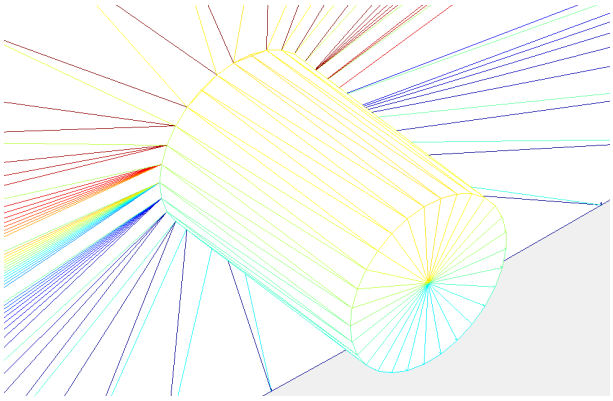


Figure 18: Fin-to-tail connector.

the cylinder function in Matlab. This can be done by creating an array for both x and y values, containing two identical columns, which define the shape of the cross section. An array of the same size is created for z values, containing one column of zeros and another of ones. By doing so, Matlab plots the desired x and y values at both 0 and 1 along the z axis, thus creating an extruded cross section. The same principle was applied in order to create the necessary surface for the tail wing of the UAV. As will be detailed later on, it is of uppermost importance to define the variables correctly in order for the surf function to correctly interpret the desired surface.
In order to create the coordinates of an airfoil, a user shared function called naca4gen was downloaded. This function allows for the upper and lower surface coordinates to be created for any 4 digit NACA designation. This function acted as the building block upon which the rest of the tail wing function was to be built. As the tail wing does not provide lift, but rather stability, only symmetrical NACA designations will be selectable by the user.
Once the chord length is specified as a function of the UAVs main wingspan, the upper and lower surfaces are defined and vertically concatenated into one array, as can be seen below. The resulting arrays are multiplied by the chord length in order to be scaled depending on user inputs, as the chord length is a function of the user specified wing span.

```
%Defining chord length of the airfoil as a function of the UAV's
    main wingspan
chord_length = wing_span/16;

%Defining upper and lower surfaces for the airfoil, for x and z
x_upper=(af.xU);
x_lower=(af.xL);

z_upper=(af.zU);
z_lower=(af.zL);

%Vertically catenating upper and lower surfaces into a single
   array,
%multiplying by chord length to scale
x_u_l=vertcat( x_upper, x_lower ).*chord_length;
z_u_l=vertcat( z_upper, z_lower ).*chord_length;
```

In order to specify the width of the aerofoil, arrays of solely ones and zeros were created, of the same size as those containing the points for the upper and lower surfaces of the x and z coordinates. The array containing only ones was then multiplied by the user defined elevator span in order to give the airfoil its desired width.

```
base=zeros(2*length(af.xU), 1);
top=(ones(2*length(af.xU), 1))*elevator_span;


y_all=horzcat(base, top);
x_all=horzcat(x_u_l, x_u_l);
z_all=horzcat (z_u_l, z_u_l);
```

Finally, the arrays of ones and zeroes were horizontally catenated giving the final array configuration for y. As well as this, the arrays containing x and z were catenated with themselves, creating an array with two identical columns. By doing so, x,y and z were defined in the same way as the cylinder function, allowing the surf function to correctly create the desired surface, as can be seen in Figure 19.

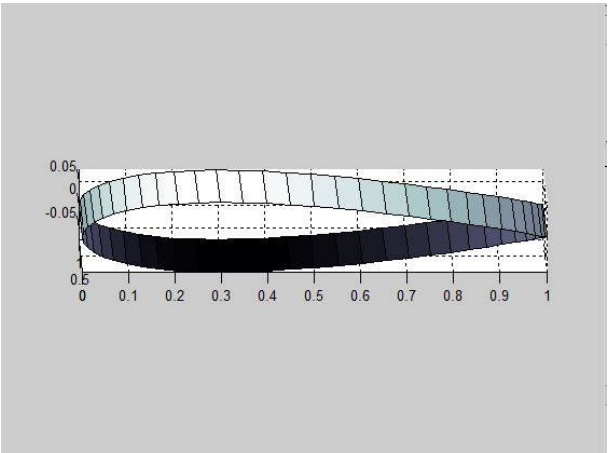


Figure 19: Initial Surface Created

However, even though this process correctly yielded the desired surface, it does not create a finite volume. In order to do so, a further level would need to be added to each of the arrays, to specify where the surface should start and finish. Initially, it was believed that the best method to achieve this would be to plot the camber line of the airfoil, as such:

```
x_camber=(vertcat(af.xC, af.xC);
z_camber=(vertcat(af.zC, af.zC);

y_all=horzcat(zeroes, zeroes, oness, oness);
x_all=horzcat(x_camber, x_u_l, x_u_l, x_camber);
z_all=horzcat (z_camber, z_u_l, z_u_l, z_camber);
```

Even though this method created a finite volume, it also lead to the creation of a non-symmetrical distribution of the patches created by the surf function, as seen in Figure 20). The result were incorrectly defined faces whose normal pointed into the volume of the surface, thus creating an unprintable .stl file. This issue was only made clear once importing the .stl file into the MakerBot software, where the part would appear in dark grey shade, and highlighted the need to correctly specify the variables to which the surf

function was to be applied. Furthermore, this method would make it much more difficult to add a cylinder (to act as a male connector peg) onto the existing airfoil, as will be detailed later.

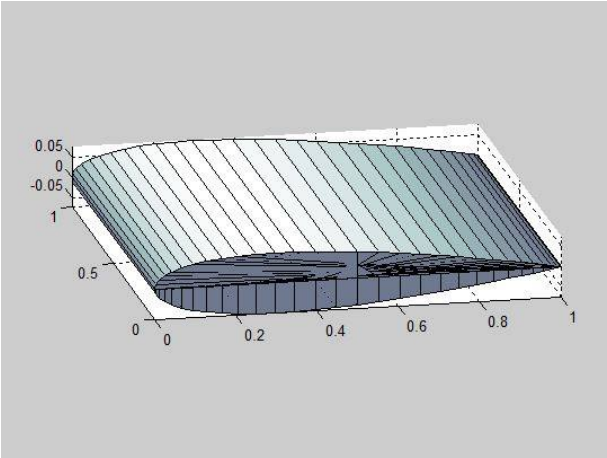


Figure 20: Incorrectly Defined Surfaces

Following this failed method, it was decided that it would be convenient to have the surfaces stem from a single point on the side surfaces. By defining this point to be the leading edge stagnation point, i.e (0,0), a symmetrical distribution of patches was achieved, yielding an .stl whose faces were correctly defined and was therefore exportable into MakerBot, as seen in Figure 21. The last step to achieving the final desired shape

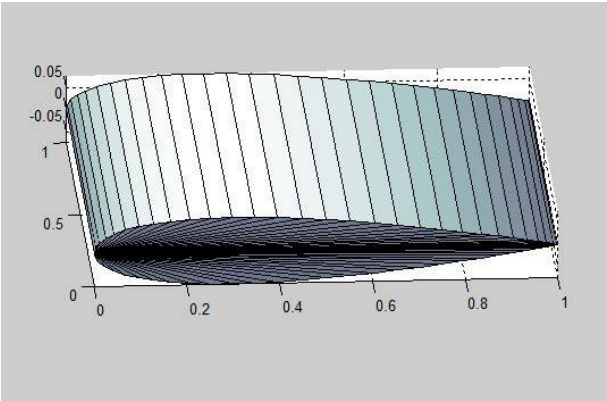


Figure 21: Correctly Defined Surfaces

would be to add a cylinder on one of the side surfaces, so as to act as a male connector peg. This would be connected to the tail wing, which in turn connects to the fuselage. In order to do so, it was determined that the single point from which the surfaces should stem should be positioned where the connector peg was to be positioned too. This offset was positioned one third up the chord length of the airfoil. Furthermore, it was necessary to specify that the number of points the cylinder should have should be the same as the number of points as the airfoil in order for there to be a one to one ratio, and therefore allow for the correct distribution of faces.

```
offset = chord_length/3;
%Defining cylinder with same number of points as airfoil

```

```
num_points=length(x_u_l);
[x,y,z] = cylinder(wing_span*0.0015,num_points-1)
```

Finally, after clearing the second row created by the cylinder function and transposing it into a column to match the array configuration, the variable `adjusted x` was created. This variable simply adjusts the x coordinate with respect to the offset defined earlier. Having done this, another level could be added to the arrays, and by carefully defining it, the cylinder was positioned as desired, and was given a length of 0.3 of the airfoils width. In order to close the cylinder and give it a finite volume, the surfaces were defined so that they ended at the cylinders midpoint, as such:

```
%Clearing first row and flipping row to column (transposing)
x(1,:)=[];
y(1,:)=[];
z(1,:)=[];
x=x';
y=y';
z=z';
%Defining adjusted x value for peg and creating offset array of
   same size as rest of variables
adjusted_x = x + offset;
offset = offset*ones(size(x));
%Defining all the "levels" for x,y,z in specific order to allow
   correct use of surf function
y_all=horzcat(base, base, top, top, top, (1.3*top), (1.3*top)
   );
x_all=horzcat(offset, x_u_l, x_u_l, offset, adjusted_x,
   adjusted_x, offset);
z_all=horzcat(base, z_u_l, z_u_l, base, y, y, y);
```

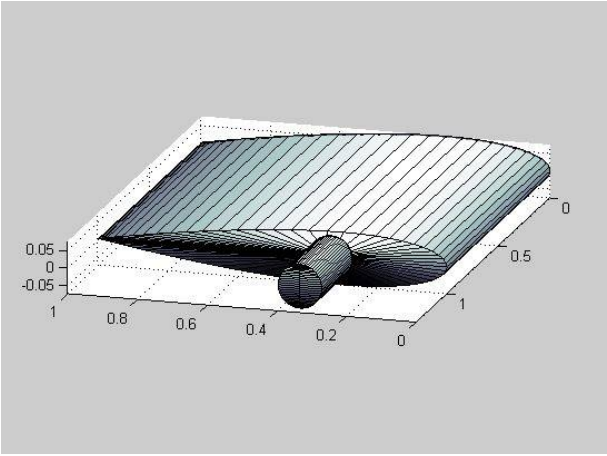


Figure 22: Tail Wing and Male Connector Peg

```
z_all = z_all - 0.0575*wing_span;
x_all = x_all + wing_span*0.3491;
if strcmp(side, 'left')
    y_all = y_all - wing_span*1.3/30 - elevator_span;
else
```

```
6         y_all = -y_all + wing_span*1.3/30 + elevator_span;
7     end
8 end
```

As can be seen in figure 22 , the part was successfully created. The closing lines of code are to do with the part's positioning in the assembly, and have an if conditional statement which allows for the peg to be positioned on either side, allowing for two mirrored parts to be created. The final part can be seen correctly into MakerBot in Figure 23.

Overall, this is a simple code that deals with the task in hand efficiently. As opposed to

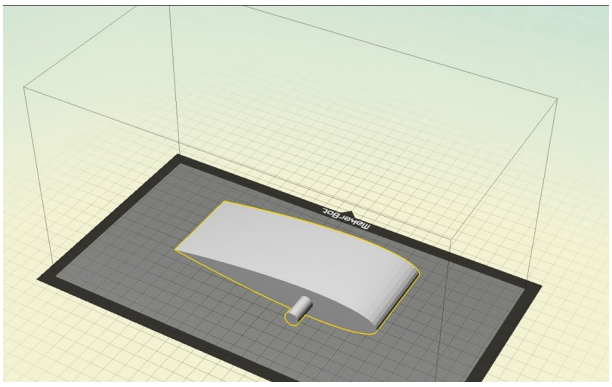


Figure 23: Correctly Imported Part in MakerBot

other more complicated parts of the UAV whose properties and final shape could vary massively, this part remains fairly similar, and is just essentially scaled according to the users input. It is for this reason that the triangulation method, detailed extensively in this report, was not necessary for this part. As was seen, if taking care when defining the surfaces to be created, a simple yet successful code was achieved. Nonetheless, it is possible that a simpler if statement could be implemented in order to deal with the mirroring issue. This could be a possible improvement for future versions of this code.

5 Assembly and Testing

The assembly of the components was straightforward - all that was necessary was to translate each part by some value in x, y, and z such that the assembly looked approximately correct. This translation had no influence upon the resulting .stl files.

The initial dimensions of the UAV were outlined in a CAD model at the start of the project (see figure 24). Unfortunately, the dimensional information in this model was poorly communicated and subsequently forgotten about. In hindsight, the dimensioning of parts without the use of hard-coded values would have been possible had we attempted to assemble the components from the beginning, and encouraged team members to use the CAD model. The hard-coded values make the code less readable, and the overall solution less elegant.

The GUI uses three structures to pass data around - UAV, Dimensions, and Graphic. These are used so that lots of information can be passed succinctly into functions. In summary of these structures:

- UAV contains the vertex and face data for each part.
- Dimension contains the user inputs from the GUI.

Figure 24: The CAD model created at the beginning of the project was intended to make dimensioning consistent, but was not used.

- Graphic holds all of the handles to each part's plot.

The function Parametric_UAV receives the user inputs from the GUI and generates the UAV part data as well as plotting the result on various axes throughout the GUI. Parametric_UAV is also responsible for error catching, for which regular expressions were used.
Taking one of these expressions to discuss, say

```
1    regexp(Dimensions.main_wing_NACA, '^[^A-Za-z\W]{4}$') == 1;
```

This expression works as follows: the caret (^) indicates that the expression must start with the expression in the square brackets. The expression in the square brackets specifies the first character should not be (that's what the ^ inside the brackets is) a letter from A-Z or a-z, or a non-alphanumeric character (i.e. ./? etc.), which is what the \W stands for. In other words, the string is only allowed to start with a number. The {4} indicates that the string can contain four numbers only. The closing $ sign states the expression should end with the 4-digit number too.
Another important feature of the assembly is its capacity to switch between exploded and assembled view. Figure 28 shows the assembled view, and figure 29 shows the exploded view. The underlying code for the animation is quite straightforward and executes as such:

1. Clear previous plot of parts.
2. Calculate new part velocity from the current time.
3. Calculate new part position based on this velocity.
4. Plot the part in its new position.
5. Pause the program to allow the animation to appear to be running in real-time.
6. Repeat.

5.1 Graphical User Interface

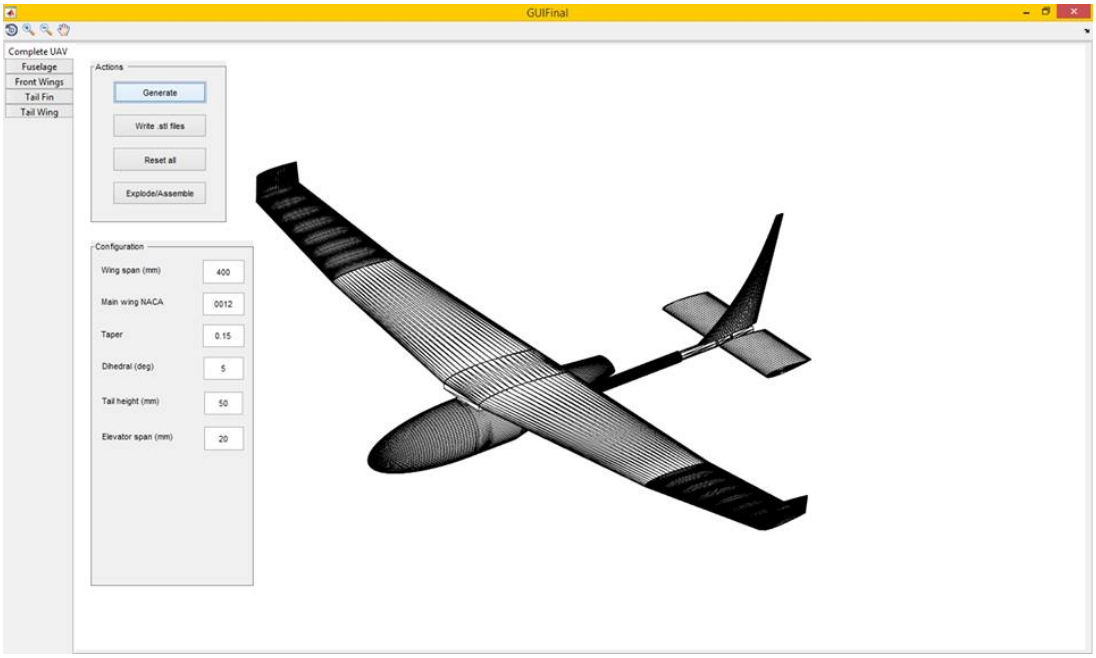


Figure 25: This figure shows the completed GUI and assembled UAV.

The Graphical User Interface, or GUI, sits on top of the code and allows the user to interact with it in a way that requires no knowledge of Matlab.
A GUI allows for applications such as push buttons, menus and toolbars to be created, meaning automated tasks can be carried out by a user who can simply change variables and click buttons to tweak an entire piece of code to their requirements.
The GUI created to display the UAV has a simple design shown in figure 25. Tabs have been created to ensure a clear layout of the GUI and enable the user to see individual components of the UAV in separately stored sections, for ease of use.

```
%Create tab group
handles.tgroup = uitabgroup('Parent', handles.figure1,'
    TabLocation', 'left');
handles.tab1 = uitab('Parent', handles.tgroup, 'Title', '
    Complete UAV');
handles.tab2 = uitab('Parent', handles.tgroup, 'Title', '
    Fuselage');
handles.tab3 = uitab('Parent', handles.tgroup, 'Title', 'Front
    Wings');
handles.tab4 = uitab('Parent', handles.tgroup, 'Title', 'Tail
    Fin');
handles.tab5 = uitab('Parent', handles.tgroup, 'Title', 'Tail
    Wing');
```

The above code creates tabs inside of the GUI interface and positions them on the left hand side of the main body of the GUI. Each progressive tab created needs to be numbered n+1 to the tab before it and can be titled depending on the section created.

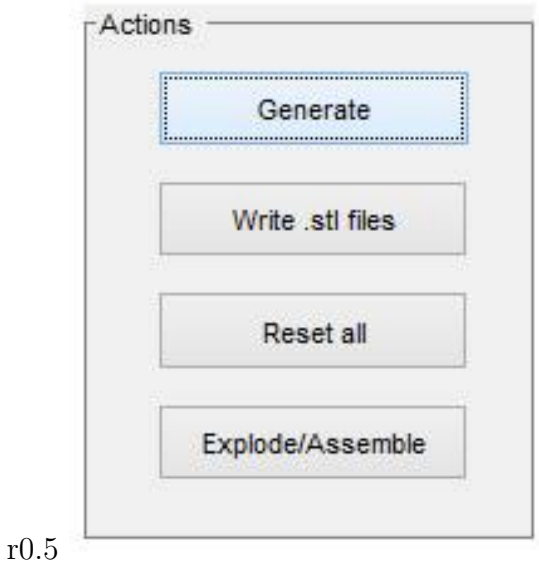


Figure 26: This figure shows the button group present on the GUI

```
%Place panels into each tab
set(handles.P1,'Parent',handles.tab1);
set(handles.P2,'Parent',handles.tab2);
set(handles.P3,'Parent',handles.tab3);
set(handles.P4,'Parent',handles.tab4);
set(handles.P5,'Parent',handles.tab5);
```

Inside the GUI figure window, panels are used to create each tab, with each separate panel holding the content of each tab. Panels are tagged P1--P5 and then associated with each tab using the handles above. The desired content is then placed on each of the panels. For the UAVs GUI, the first tab holds the entirety of the content and is where the user will spend most of their time. On the first tab, axes act as the main body, with static text, editable text and various buttons allowing the user to control certain aspects of the program. All subsequent tabs have only an axes implemented for the purpose of viewing specific components of the UAV.
The panel for P1 has to be equal to or greater than in size to that of the other tabs created, to ensure that the program size remains constant. Also, inside of the GUI figure window, panel P1 must standalone and not be overlapped by any of the other tabs, else this will show on the GUI interface.

```
%Reposition each panel to same location as panel 1
set(handles.P2,'position',get(handles.P1,'position'));
set(handles.P3,'position',get(handles.P1,'position'));
set(handles.P4,'position',get(handles.P1,'position'));
set(handles.P5,'position',get(handles.P1,'position'));
```

All panels are then repositioned to the same location as P1, so that flicking through the tabs causes panels to overlap each other in the same space.

There are four buttons on the GUI which are used to execute various operations. Buttons operate by executing the code placed inside their specific function when pressed. All buttons are placed inside a button group, shown in figure 26, and positioned to the

left of the UAV image.
The first button is a push button named Generate which runs the main bulk of the code written in another .m file, to generate the UAV images and display them onto their corresponding axes throughout the GUI, using the values inserted into the various text boxes.
The second button simply writes and saves the .stl files for the UAV shape generated. For shapes that were surfed, the surf2stl command is used, for every other shape generated, stlwrite is utilised. After writing the files, the button also creates a pop-up message to the user stating that the files have been made. Below is an example of two .stl write commands under the write .stl button function.

```
function CompGenSTLPB_Callback(hObject, eventdata, handles)
stlwrite('UAV_tail_fin.stl', UAV.tail_fin.T, [UAV.tail_fin.x
    UAV.tail_fin.y
UAV.tail_fin.z]);
surf2stl('UAV_tail_wing_left.stl', UAV.tail_wing_left.x, UAV
    .tail_wing_left.y,
UAV.tail_wing_left.z);
msgbox('.stl files successfully written.');
```

The third button, Reset all, simply clears all the axes in each tab using the cla command shown in the example below, effectively resetting the program.

```
function clear_button_Callback(hObject, eventdata, handles)
axes(handles.Assembly_axes);
cla
axes(handles.front_wings_axes);
cla
```

The fourth button is a toggle button which executes an animation of the UAV in an exploded view. Clicking the button again returns the image back to its previous, assembled state. This is achieved by using an 'if' and 'else' statement, which takes the two states in which the toggle button can be, 0 and 1, and assigns the corresponding animation to each state.

```
function explode_button_Callback(hObject, eventdata, handles)
wing_span = str2double(char(get(handles.wing_span_input,
    'String')));
assignin('base', 'wing_span', wing_span);
assignin('base', 'handles', handles);
axes(handles.Assembly_axes);

if evalin('base', 'view_mode') == 0
    evalin('base', '[UAV Graphic] = explosion_animation(
        UAV, Graphic, wing_span, 1, handles);');
    view_mode = 1;
else
    evalin('base', '[UAV Graphic]= explosion_animation(UAV
        , Graphic, wing_span, 0, handles);');
    view_mode = 0;
```

```
end
assignin( base , view_mode , view_mode);
```

Figure 27: User defined variables in the GUI

Inside the configuration panel shown in figure 27 lies the user changeable data which directly affects the UAV generated. Each of the six variables are shown with pre-chosen default values and are linked to the variables, for each function, with the code shown below. This piece of code collects the data stored into each text box and points them to replace their respective variables, allowing the generation of the desired UAV design.

```
function generate_button_Callback(hObject, eventdata, handles)
% Capture parameter inputs from GUI
Dimensions = struct;
Dimensions.wing_span = str2double(char(get(handles.
    wing_span_input,'String')));
Dimensions.main_wing_NACA = get(handles.main_wing_NACA_input, '
    String');
Dimensions.taper = str2double(get(handles.taper_input, 'String')
    );
     Dimensions.taper = Dimensions.taper(1);
Dimensions.dihedral = str2double(get(handles.dihedral_input, '
    String'));
Dimensions.tail_height = str2double(get(handles.
    tail_height_input, 'String'));
Dimensions.elevator_span = str2double(get(handles.
    elevator_span_input, 'String'));

[UAV,Graphic] = Parametric_UAV(Dimensions, handles);
```

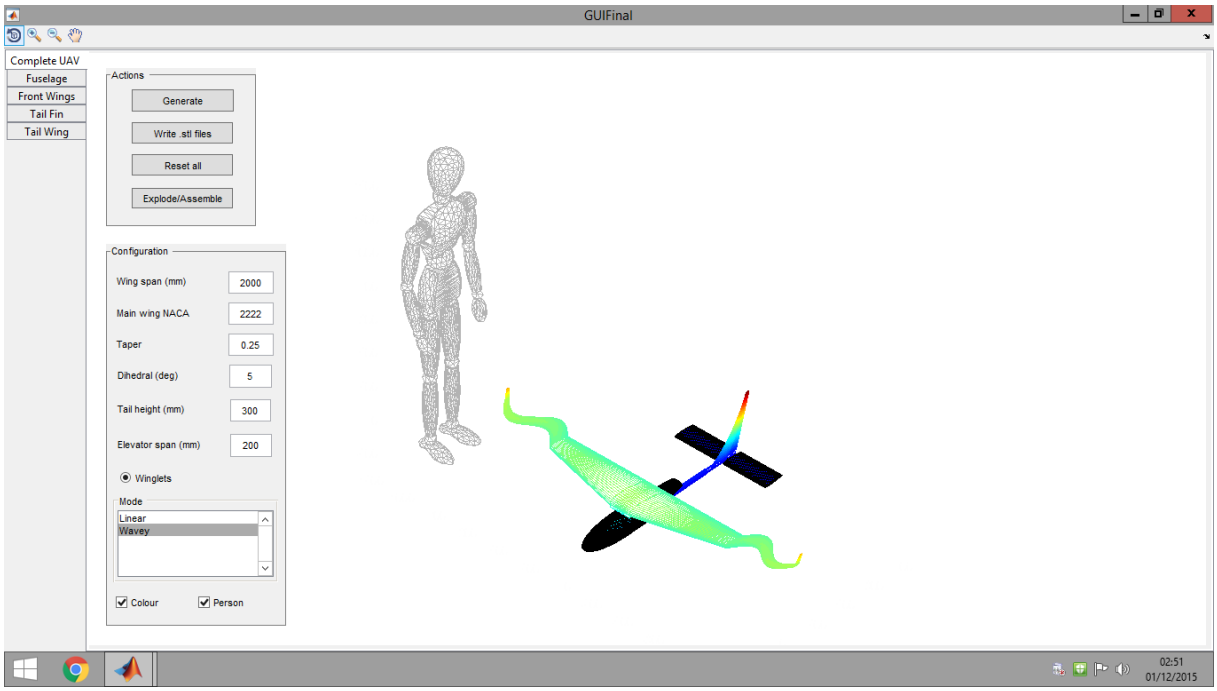


Figure 28: Assembled configuration of UAV.

```
assignin('base', 'UAV', UAV);
assignin('base', 'Graphic', Graphic);
```

The final noteworthy feature of the GUI/assembly is its ability to set the taper mode of the wings. This can be selected using either the listbox, or by typing in a function. In both cases, the mode/function is captured from the GUI as a string, and is then passed through to the wing_tip function. The wing_tip function then determines whether a function has been entered or a mode selected from the listbox. If a mode is selected, the relevant conditional statement is executed. If a valid function has instead been entered, it's executed using the eval() function. The code below shows how this taper-mode selection works.

```
if regexp(Dimensions.winglet_function, '^[k(\s)=].+') == 1;
    start_of_expression = regexp(Dimensions.winglet_function, '=
        ');
    expression_input = Dimensions.winglet_function((
        start_of_expression+1):end)
    x_function_trans_factor = eval(expression_input);
else
    if strcmp('Wavey', Dimensions.winglet_mode)
        full_circle_y = 2.*pi.*y./max(y); % Maps the y values
            between 0 and 2 pi
        x_function_trans_factor = sin(full_circle_y)*Dimensions.
            wing_span./32;
    elseif strcmp('Linear', Dimensions.winglet_mode)
    elseif strcmp('Quadratic', Dimensions.winglet_mode)
        n_y = y/max(y);
        x_function_trans_factor = (n_y.^2)*Dimensions.wing_span
            ./16;
```

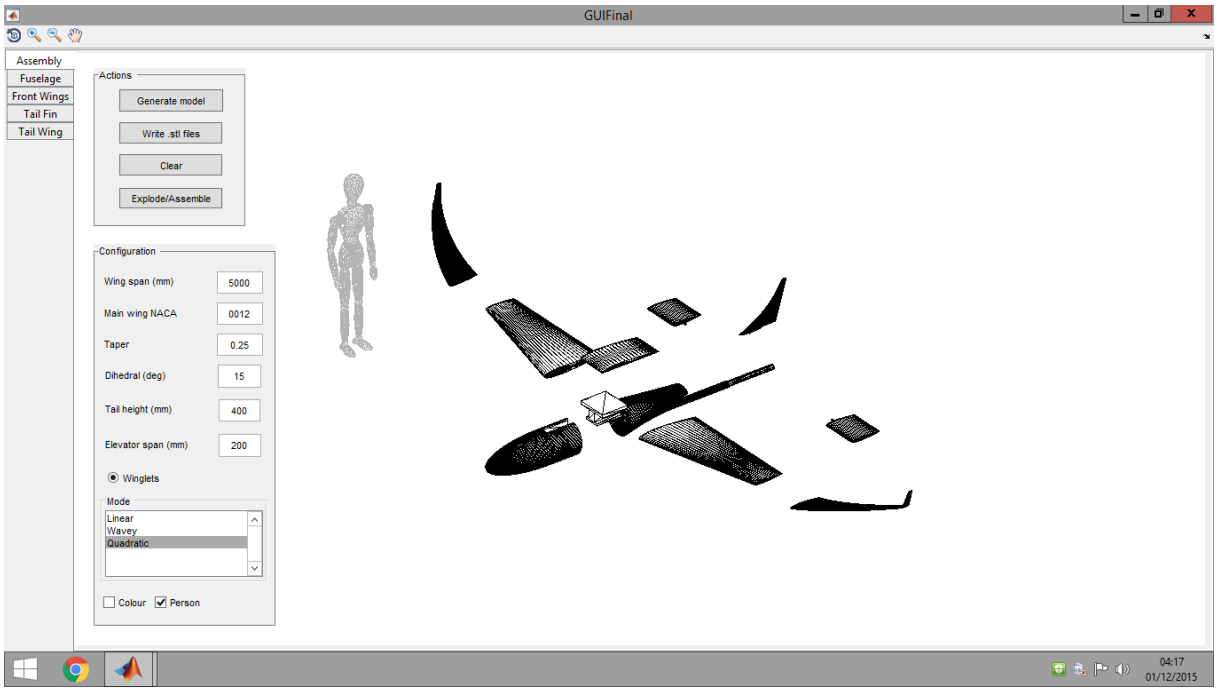


Figure 29: Exploded configuration of UAV.

```
13        end
14    end
```

This feature of the GUI isn't safe, as it allows the user to execute any function they wish. A minor precaution against this was taken by mandating that the expression begins $k =$, but this could easily be bypassed by first assigning k then writing another line, i.e.

```
1         k = 1;
```

6 Process Evaluation

As mentioned in the introduction, the group's approach to this project was fractured; it resulted in vast quantities of time being spent refactoring one another's code so that the parts would fit together, take the correct number of arguments, etc. Communication was generally poor between members of the team, and it was infrequently clear exactly what the state of each member's function was. Part of the reason for this was the team size, which hindered the ability of the team to respond quickly to testing feedback or edits to a particular function. Another contributing factor to our organisational inefficiency was our content management system - Google Drive is poorly suited to software development. It would have been desirable to use Git, however there was insufficient time for group members to become familiar enough with the system to use it effectively.
In spite of the team's imperfections, the project was ultimately successful. As the sense of urgency increased, so too did the levels of communication and efficiency. The result speaks for itself - a clean, simple-to-use interface with enough depth to stimulate users without overwhelming them, and underlying code that is generally well-commented. Moreover, each component was shown to be printable, and each member gained a considerably better understanding of Matlab. Not only this, but they did so independently - starting by playing with simple functions, moving through to writing moderately complex functions

that contained error-catching.
An essential improvement to the team's approach to the project would be to designate a coordinator, whom would be responsible for the assembly throughout the project. In addition to this, they should liaise with each team member to ensure that there the team shared a common vision.
The practical skills gained in this project should also not be forgotten - many team members learned how to use a 3D-printer, a skill that is useful for prototyping simple mechanical designs.

7 Conclusion

Our finished code performs well and allows for a large amount of flexibility. A graphical user interface is produced that allows the user a significant amount of freedom to generate and customise printable UAVs of all shapes and sizes! If there were more time to improve the design, more focus would have been taken to ensure the aerodynamics would consistently create enough lift for the UAV to fly well and for enough time and distance to complete a task. Another possible considerations for future development would be to alert the operator if and when they have exceeded the possible print size for a certain 3D printer model, i.e. Makerbot Replicator 2. Following from this concept, disecting the components of the UAV when their respective sizes exceed the printer capabilities would significantly increase the codes potential in industry.

Overall the code is clear and user friendly and the GUI is easy to use for operators with no programming experience and allows a wide range of options in the design of the UAV.

References

[1] Wikipedia. Wing configuration, . URL `https://en.wikipedia.org/wiki/Elliptical_wing`.

[2] JAD MOUAWAD. How winglets work, October 2013. URL `http://www.nytimes.com/interactive/2013/10/24/business/Why-Winglets.html?_r=1&`.

[3] George C. Larson. How things work: Winglets, September 2001. URL `http://www.airspacemag.com/flight-today/how-things-work-winglets-2468375/?no-ist`.

[4] Wikipedia. Wingtip device, . URL `https://en.wikipedia.org/wiki/Wingtip_device`.