Design of Simple Aircraft and Visualization of Fluid Velocity and Vorticity

Gregory Devenport

A capstone report submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Andrew Ning, Advisor

Department of Physics and Astronomy

Brigham Young University

ABSTRACT

Design of Simple Aircraft and
Visualization of Fluid Velocity and Vorticity

Gregory Devenport
Department of Physics and Astronomy, BYU
Bachelor of Science

The design process for a simple radio controlled (RC) airplane with high lift is presented. The lift-to-drag ratio of the designed aircraft is measured experimentally and is compared to the lift-to-drag ratio calculated using computational tools Xflr5, and FLOWUnsteady [1]. To augment FLOWUnsteady simulations of a wind harvesting aircraft, a successful coding method is presented to allow visualization of the vorticity and velocity of air near the turbine blades.

Keywords: glide ratio, isosurface, FLOWUnsteady, vorticity

ACKNOWLEDGMENTS

I want to thank Adam Cardoza for mentoring me during my airplane design project, Judd Mehr for mentoring me while I did research under him, and Dr. Andrew Ning and the BYU FLOW Lab for helping me learn more about aerodynamics and coding. A special thanks to my daughter Ashton for assisting me in building my RC plane and most of all to my wife Alyson for helping me on my entire journey at BYU and through life.

Contents

Table of Contents **iv**

1 Introduction **1**

2 Aircraft Design Project **2**
2.1 Design . . . 3
2.1.1 Airfoil Shape . . . 3
2.1.2 Wing Design . . . 5
2.1.3 Plane Dimensions . . . 7
2.2 Analysis . . . 7
2.3 Flight . . . 10

3 Flow Visualization Using FLOWUnsteady **12**
3.1 Coding Method . . . 14
3.2 Validation of Code . . . 15
3.3 Windcraft Curved Path . . . 17
3.4 Documentation . . . 18

4 Conclusion **19**

Appendix A Aircraft Videos **20**

Appendix B Visualization Code **21**

Appendix C Isosurface Creation Guide **30**

References **33**

Chapter 1

Introduction

The use of aircraft is critical to our lives today. Aircraft are used for a variety of purposes including passenger and cargo transport, defense, advertising, and recreation. Current research seeks to expand the use of aircraft to areas such as urban transportation, and even wind energy harvesting [2].

This report examines some of the basic aerodynamic principles that affect the take off and landing distance of an airplane and details the process to design a simple RC airplane with a short take off and landing distance. It also describes an analysis of the aircraft using computational tools Xflr5 and FLOWUnsteady, and compares these computational results to experimental data for the same aircraft.

This report also outlines a research opportunity working with a PhD student on wind turbine placement optimization for the Makani M600 Energy Kite. Code was developed to aid in viewing flow characteristics around the windcraft to help with turbine placement optimization and to create images for use in papers.

Chapter 2

Aircraft Design Project

Many factors influence the takeoff and landing distance of an airplane, but two significant factors are the aircraft weight and the ratio of the lift to drag forces on the aircraft. While the original design goal for the project was to minimize takeoff and landing distance, it was decided that experimentally measuring this would introduce non-aerodynamic factors such as runway composition and pilot skill. To place more emphasis on aerodynamics, the design goal was modified to maximize the lift to drag ratio, or glide ratio. Increasing lift lowers the stall speed of the aircraft. This means the airplane can reach takeoff speed sooner, therefore reducing the takeoff distance. With a lowered stall speed, the aircraft can also touch down at a slower speed which reduces the landing roll of the aircraft.

The design process began with identifying several characteristics that would contribute to a high glide ratio such as airfoil shape, wing span and sizing, and aircraft weight. These characteristics are described below.

2.1 Design

2.1.1 Airfoil Shape

To increase lift, an airfoil with camber and a large chord length is desirable. High lift airfoils are usually thin and long with a lot of camber. Airfoils were designed in two batches. The first batch were designed with the only goal being to maximize the airfoil coefficient of lift, c_l. One of the first designs in shown in figure 2.1.

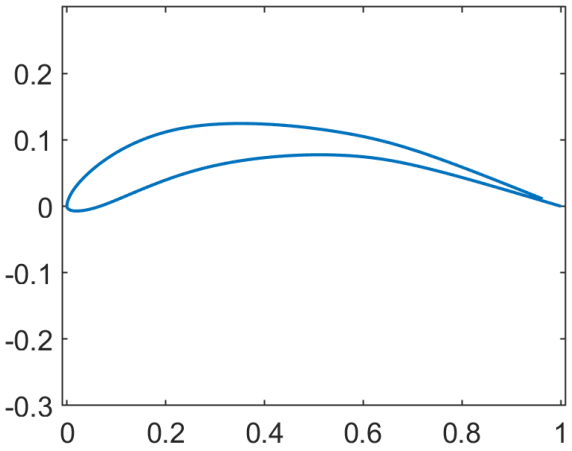


Figure 2.1 First round airfoil profile.

The general results of these first airfoils were large lift coefficients, but the pitching moment coefficient values, c_m were significantly greater than for commonly used airfoils. Figure 2.2 shows the c_l and c_m plots corresponding to the airfoil in figure 2.1. The RC plane was to be constructed from insulation foam so a thin airfoil also presented structural concerns. Ultimately these airfoils were abandoned due to the stability and structural concerns. The second batch of airfoils were modeled after Clark Y and sailplane foils. The Clark Y airfoil is commonly used in RC planes, and sailplanes are designed to stay in the air for a long time without power so they have very large glide ratios. These characteristics made both airfoils a good starting place for airfoil design. The general shapes of these airfoils were modified until it seemed that the airfoil was stable but also

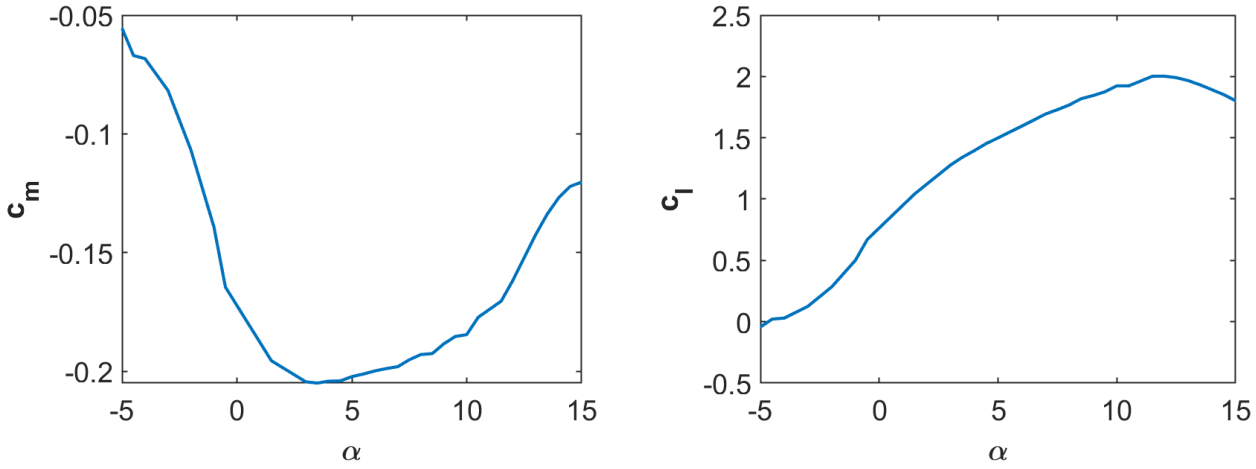


Figure 2.2 First round airfoil c_l and c_m.

produced a lot of lift. The airfoil that was ultimately selected is shown in figure 2.3 with its c_m and c_l plots shown in figure 2.4. Its positive camber gives a large amount of lift, but the airfoil remains relatively stable. Its thickness also allows it to be cut from foam.

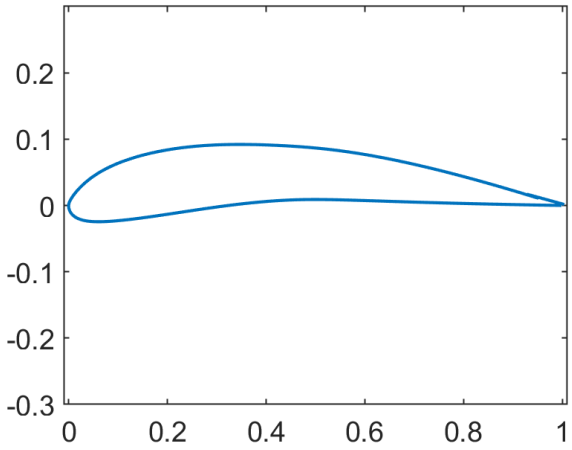


Figure 2.3 Second round airfoil profile.

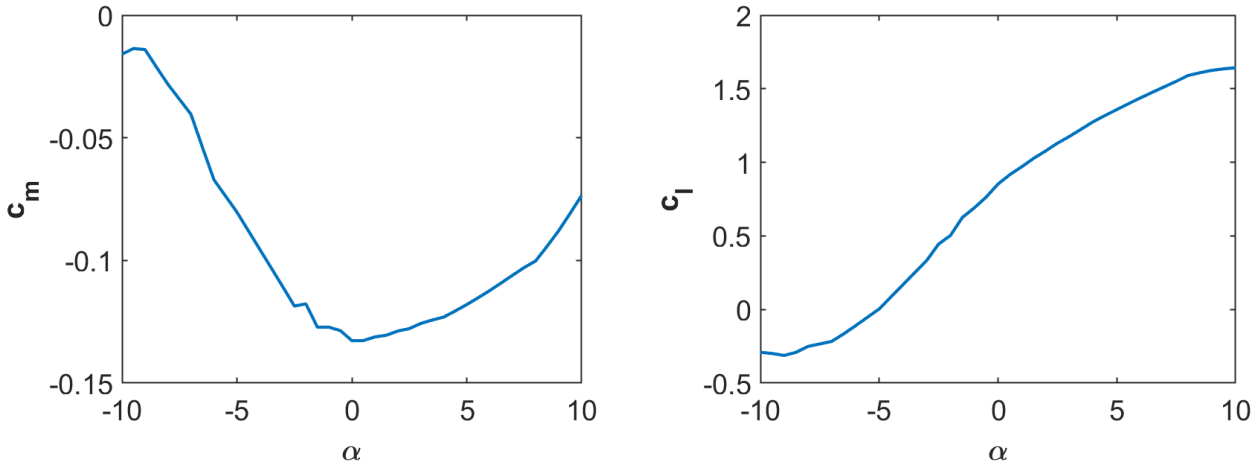


Figure 2.4 Second round c_l and c_m.

2.1.2 Wing Design

To increase the glide ratio, a large wingspan was desirable. This decreases induced drag while increasing lift. However, a large wingspan poses a structural problem especially for a wing built of foam as the bending moment at the root can be large. A wingspan was chosen that would provide a lot of lift while not breaking at the root due to bending stress. The original wing design had a root chord of $9.5cm$ but this was doubled to make the wing more rigid for construction. This increased the wing area which was found to increase lift and lower the stall speed of the aircraft despite a small increase in drag. Doubling the chord length also increased the wing thickness which decreased the bending stress in the wing.

To improve the lateral stability of the aircraft, the wing was given 2° dihedral. This remained constant in the computational analyses but the aircraft that was built had a highly flexible wing that bent significantly when in flight. This increased the effective dihedral as shown in figure 2.5. Increasing the dihedral caused the plane to be very laterally stable, but lowered the glide ratio, as the vertical component of lift was decreased and the drag was increased.

The main wing was twisted at the tip so the control surfaces wouldn't stall before the inboard

Figure 2.5 Extra dihedral in wing while in flight.

wing, allowing for control even during a stall. The entire wing was also twisted to increase the glide ratio at level flight. Shown in figure 2.6 is the glide ratio plotted against the angle of attack. The max glide ratio occurs around 7° so the entire wing was twisted 5° to increase the glide ratio at level flight. The wing was not twisted to the angle of attack corresponding to max glide ratio because this is near the stall angle of attack for the wing and the ability to climb by increasing the angle of attack was needed for flight.

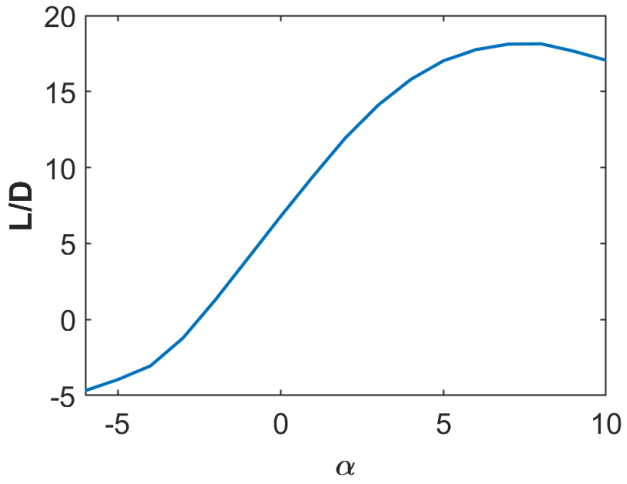


Figure 2.6 Untwisted wing L/D.

2.1.3 Plane Dimensions

The final plane design, excluding the fuselage, is shown in figure 2.7 with the various dimensions tabulated in table 2.1. No rigorous analysis was performed to select the size of the horizontal and vertical stabilizer, but stabilizers were sized according to statistical values [3]. The total area of the horizontal stabilizer was chosen to be about 20% of the main wing area. The total area of the vertical stabilizer was chosen to be about 10% of the main wing area. These values were found to be standard for rough tail sizing. The vertical and horizontal stabilizers use a NACA 0012 as their airfoil. The total mass of the plane was 515 grams and the completed plane is shown in figure 2.8 with some internal electronics not shown.

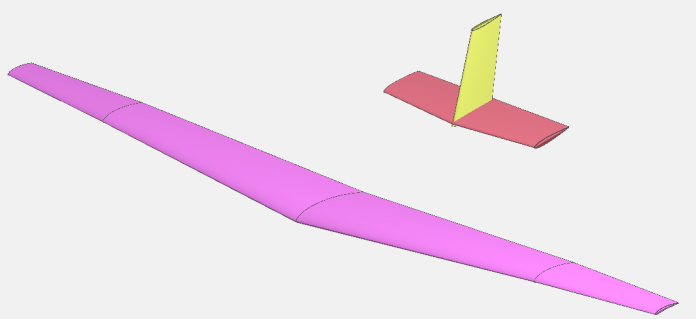


Figure 2.7 Final plane design.

2.2 Analysis

Analysis of the plane was performed using FLOWUnsteady and Xflr5. Xflr5 was used to perform stability analysis and to find the angle of attack corresponding to a high glide ratio as discussed before. FLOWUnsteady was used as a comparison against Xflr5, but was primarily used to prepare for future research, as research in the lab often involves using FLOWUnsteady. FLOWUnsteady treats a wing as a flat plate, discounting the wing's airfoil profile. So, to more accurately compare the FLOWUnsteady analysis to the Xflr5 analysis, a second Xflr5 analysis was performed with all

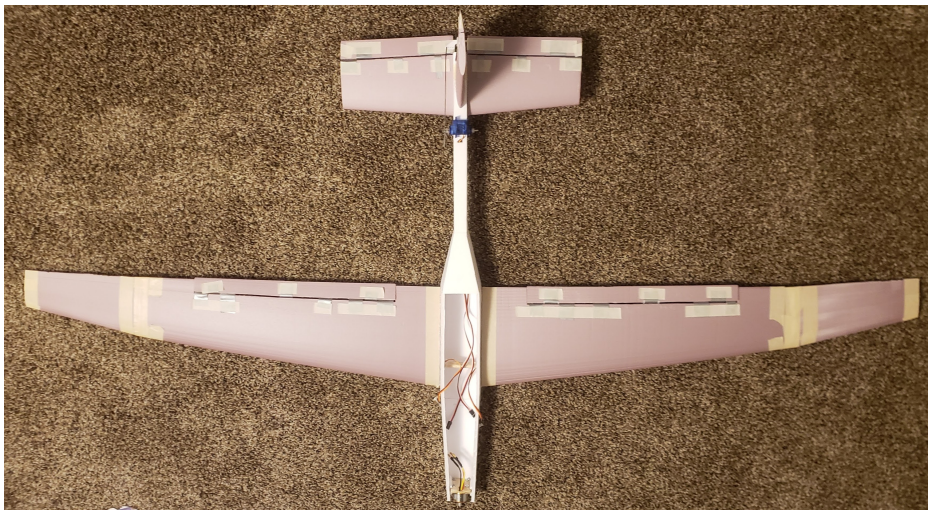


Figure 2.8 Finished RC plane.

	Main Wing	**Horizontal Stabilizer**	**Vertical Stabilizer**
Span	150 cm	17 cm	24 cm
Root Chord	19 cm	10 cm	10 cm
Tip Chord	6.5 cm	8 cm	6 cm
Root Twist	5°	-1.5°	0°
Tip Twist	1°	-1.5°	0°
Sweep	7°	5°	14°

Table 2.1 Plane dimensions.

airfoils replaced by a NACA 0003 airfoil which is nearly a flat plate. Analyses were then performed to compare aircraft glide ratios. Each plane had the exact same dimensions, twist, sweep, and dihedral. The values of C_D and C_L from these analyses are shown in figure 2.9. It was found that for the three analyses the C_L data was nearly identical, with the original Xflr5 plane data slightly higher. This is to be expected as the camber of the wings increases the amount of lift produced. The C_D data for each analysis differed greatly, with the FLOWUnsteady values significantly smaller than both other analyses. The FLOWUnsteady values are on average 14% of the original Xflr5 plane

values and 11% of the Xflr5 flat plate airfoil values. The Xflr5 flat plate airfoil data is similar to the original Xflr5 plane data until at an angle of attack of 3° it increases very quickly. One reason for the FLOWUnsteady data being smaller than both Xflr5 data is that FLOWUnsteady only calculates induced drag and does not include parasitic drag due to friction and pressure. Assuming parasitic drag is roughly equal to induced drag, the data is still smaller than expected. The reason for this error is not known, but could be due to the setup of the simulation.

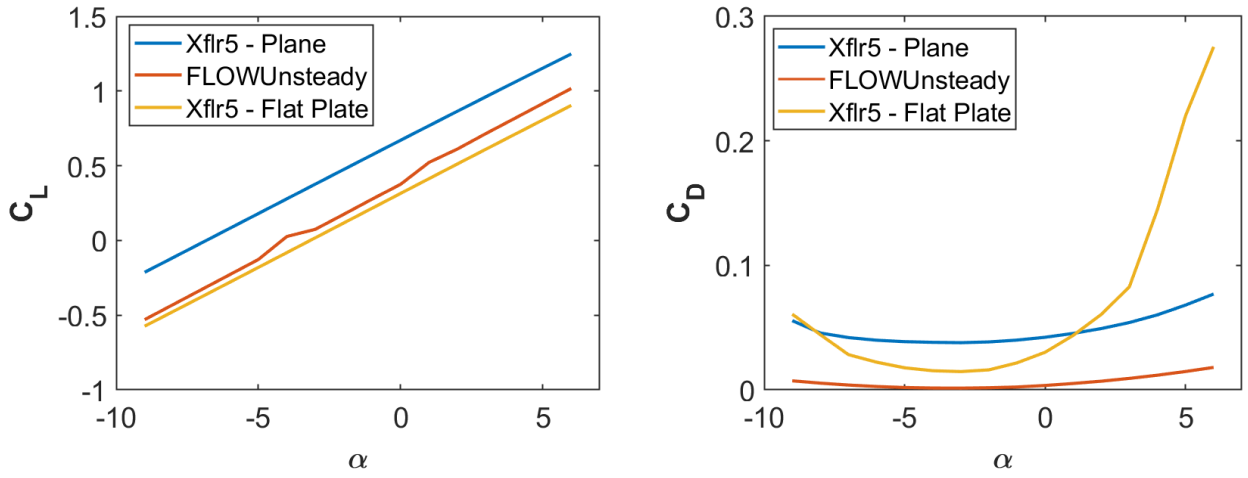


Figure 2.9 C_L and C_D comparison.

Based on these analyses FLOWUnsteady predicted the glide ratio to be 120.1, the Xflr5 flat plate analysis predicted a glide ratio of 10, and Xflr5 analysis using the designed airfoil predicted a glide ratio of 18.25. The FLOWUnsteady prediction is high, likely due to FLOWUnsteady not including parasitic drag as discussed before. Using the previous assumption relating induced and parasitic drag, FLOWUnsteady with estimated C_D corrections predicts a glide ratio of 60 which is still extremely high. Comparing these predictions to the experimental data shown below, the analysis in Xflr5 using the designed foil was the most accurate.

Throwing Height(ft)	Glide Distance (ft)	Glide Ratio
6	148	24.7 ± 2.1
6	172	28.7 ± 2.4
6	113	18.8 ± 1.6
6	142	23.7 ± 2.0

Table 2.2 Experimental glide ratio.

2.3 Flight

The glide ratio can be determined experimentally by measuring the horizontal distance traveled, x, and the corresponding loss of altitude, y. As shown in equation 2.1 the glide ratio is found by dividing the horizontal distance traveled by the altitude loss.

$$\frac{L}{D} = \frac{x}{y} \tag{2.1}$$

Experimental data was taken at Rock Canyon park. The plane was thrown from a measured height of 6 ft and the horizontal distance traveled before touching the ground was measured. The collected data from four test throws is shown in table 2.2. The glide ratio was determined by using equation 2.1. The uncertainty, Δy, in the throwing height was estimated to be 0.5 ft, and the uncertainty, Δx, in the horizontal distance traveled was estimated to be 2 ft. The total uncertainty in the glide ratio is then given by equation 2.2.

$$\sigma = \sqrt{\left(\frac{\Delta x}{y}\right)^2 + \left(\frac{x\,\Delta y}{y^2}\right)^2} \tag{2.2}$$

Averaging the experimental values, the glide ratio of the plane is 24.0 $\pm$2.0 which is higher than the predicted values from both Xflr5 analyses, and significantly lower than the predicted value from FLOWUnsteady. While the FLOWUnsteady data for C_L seems to be accurate, as mentioned before

the C_D data is in question. The aircraft performed well and the goal of producing an airplane with a high glide ratio was accomplished. Videos of glide tests and flights can be found in appendix A.

Chapter 3

Flow Visualization Using FLOWUnsteady

The use of FLOWUnsteady in aircraft analysis led to a research opportunity working with a PhD student on wind turbine placement optimization for the Makani M600 Energy Kite, referred to as a windcraft in this report. An image of the windcraft is shown in figure 3.1. The research involved

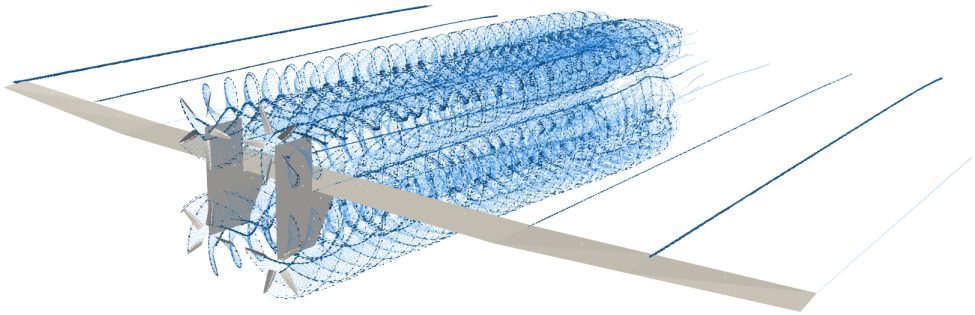


Figure 3.1 Windcraft in simulation.

using FLOWUnsteady to simulate the windcraft's behavior in flight with the purpose of optimizing

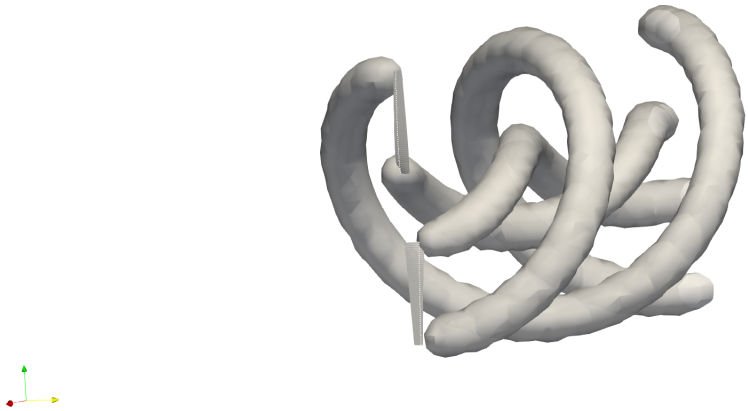


Figure 3.2 Isosurface of vorticity.

the placement of the turbines for electric power production [5]. To aid in the optimization, and to create images for future papers, the PhD student wanted to visualize the behavior of the airflow, specifically the velocity and vorticity, near the windcraft turbines. Vorticity is a measure of the rotational characteristics of a flow, and was visualized as an isosurface, which shows a surface with constant value, in this case vorticity. As a visual example, figure 3.2 shows an isosurface for a specified value of vorticity around a propeller.

A FLOWUnsteady simulation calculates velocity and vorticity, however these values are only computed at aircraft surfaces such as wings, turbine blades, and propellers. While this is useful, flow characteristics needed to be visualized at locations other than the windcraft surfaces. Code to accomplish this already existed but the existing method was not documented. This project was performed to develop and document a method for determining fluid velocity and vorticity everywhere inside a defined volume using the output of an existing FLOWUnsteady simulation. It is important to note that the purpose of the code was to produce a correct qualitative description of the flow, rather than to calculate exact numerical values of fluid characteristics, although this may be a result.

3.1 Coding Method

FLOWUnsteady uses the vortex particle method to numerically solve the Navier-Stokes equations, providing velocity and vorticity information about the flow [4]. For each simulation time step, a certain number of particles are shed from aircraft surfaces. The particles' location, vortex strength, and size are used to calculate flow characteristics.

The code was developed by solving a series of problems. After a discussion about the general process with the writer of the existing code, it was determined that the main problems to solve were recreating the simulation particle field, calculating the velocity and vorticity based on the particle field, and exporting the velocity and vorticity data in a format that could be viewed in the visualization application ParaView. After working through each of these problems, and much trial and error, a working method was developed and documented.

To calculate flow characteristics everywhere inside a defined volume, the simulation particle field must first be recreated using the output files from a FLOWUnsteady simulation. For a given time step, needed particle data is read in from the simulation output file. Each particle is recreated with a position, vortex strength, and smoothing radius. The smoothing radius defines a distance over which the vortex influence of the particle decreases so the velocity doesn't go infinite as the radial distance from the particle goes to zero. A fluid domain grid is created to which velocity and vorticity solutions will be added later. The grid will eventually become a data set with velocity and vorticity defined at regularly spaced nodes.

A particle field of test particles is created to be used as points to calculate the velocity and vorticity. These particles are placed at the same locations as the fluid domain grid nodes. For these particles, small values for the vortex strength and smoothing radius are used so the particles do not significantly affect the velocity and vorticity solutions.

A function is called to calculate the interaction between the simulation particle field and the test particle field. The use of this function was copied from the existing code. The velocity and vorticity

induced by the simulation particles on the test particles is stored in the respective test particles. By iterating through each test particle, the velocity and vorticity throughout the volume is extracted. These values are then added to the fluid domain grid which can be saved as a VTK file to be viewed in ParaView.

This process is repeated for the desired number of time steps. Simulations can be viewed in ParaView, and by importing the fluid domain grid VTK files, velocity and vorticity inside the desired volume can be viewed as the aircraft simulation is played.

3.2 Validation of Code

As previously stated, the code was developed to provide an accurate but qualitative representation of the flow. The numerical values calculated may be correct, but the development and validation were only concerned with correct general visual behavior. A few simple test cases were used to validate as well as debug the code. The first test was using a hovering propeller. The vorticity in the vicinity of a propeller should be high at the propeller tips. Figure 3.3 shows a slice of the vorticity profile. As can be seen, the vorticity is strongest at the propeller tips as expected.

The second test case used was the velocity profile behind a set of wind turbine blades. The air velocity behind turbine blades will be slower than the free stream velocity of the incoming air because the turbine extracts energy from the air, thus reducing the kinetic energy of the air. Figure 3.4 shows a slice of the velocity profile, taken slightly behind the turbine blades. The air surrounding the turbine blades is of uniform velocity, and the air directly behind the turbine blades has varying, but lower velocity.

The third test case used was a comparison of the velocity at the wing and pylon surfaces to the velocity near both surfaces. As FLOWUnsteady correctly calculates the velocity at the wing and pylon surfaces, a comparison very near these would provide a good test case. As figure 3.5 shows,

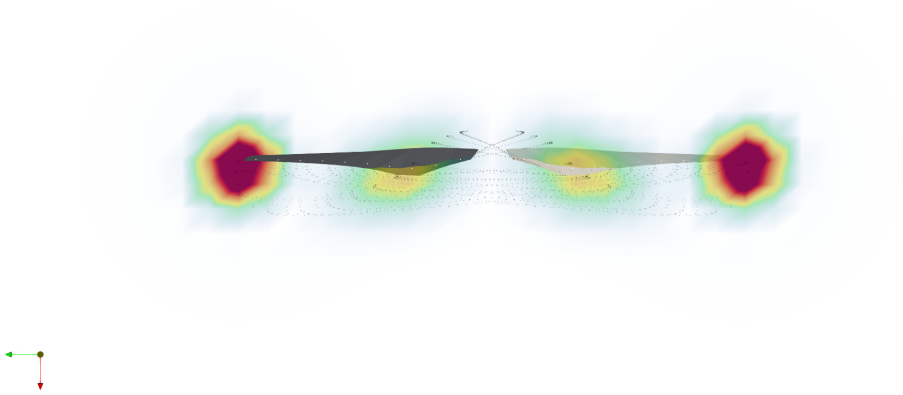


Figure 3.3 Propeller in hover.

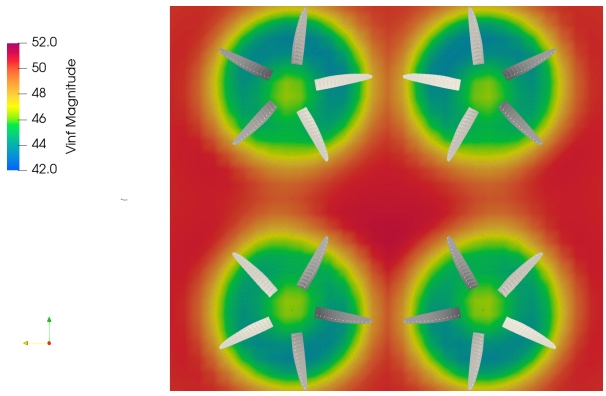


Figure 3.4 Front view of four wind turbines.

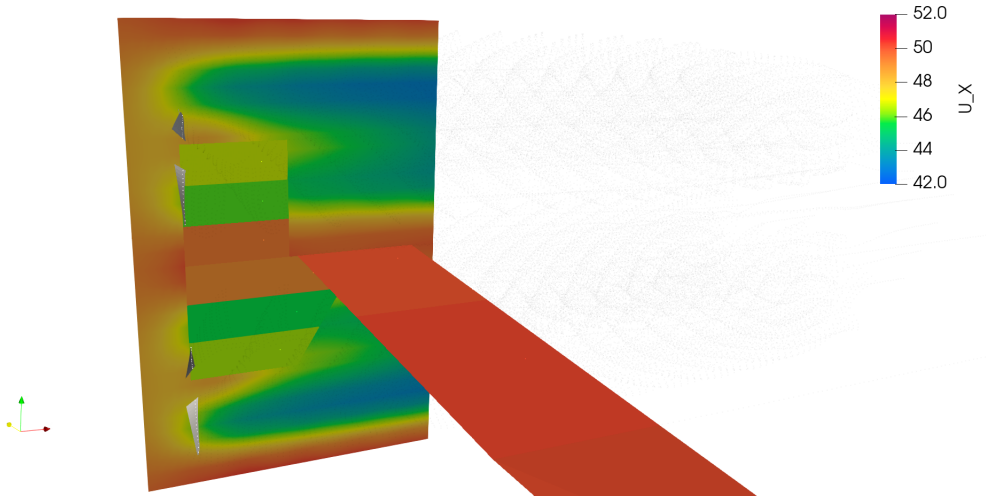


Figure 3.5 Wing and pylon velocity profile.

the velocity near the pylon and wing matches closely with the velocity at the surface of the pylon and wing. FLOWUnsteady calculates the velocity at surfaces in a specified number of panels and reports the velocity of each panel as an average of the velocity along that panel. For example, the pylon in figure 3.5 has 6 panels and the velocity of each panel is an average. This could explain the slight difference between the velocity at the surfaces and the nearby velocity.

The success of these three test cases confirmed the code was correctly modeling the flow properties and could be qualitatively used to aid in the windcraft study.

3.3 Windcraft Curved Path

The code was developed under the assumption that the windcraft was stationary, so the defined volume to calculate velocity and vorticity inside was also stationary. However, the actual movement of the windcraft is circular so the code needed to be modified so that the velocity and vorticity could be calculated as the windcraft moved. The only needed change was to move the volume in which velocity and vorticity were calculated. Using the velocity of the windcraft and the radius of its

path, the defined volume is set to move with the windcraft. The center of the calculation volume is defined by the user, and for each time step, the bounds are defined based on the location of this center.

3.4 Documentation

Following the completion of the validation cases and the added functionality for a moving calculation volume, a document was created for the PhD student describing how to use FLOWUnsteady to create velocity profiles and vorticity isosurfaces. As part of the motivation for this project was a lack of available documentation, the document helped fill this gap. It provides a step by step guide outlining the code process and using ParaView to view the results. The document is found in appendix C.

Chapter 4

Conclusion

The aircraft built had a glide ratio higher than expected, and performed well in flight. One limitation of the design project was that the various design decisions were made after only a few iterations. For example, to maximize the glide ratio, various parameters were changed such as twist, wingspan, and chord length until the glide ratio seemed to be maximized. For future work, using a computer optimization to make the various design considerations could yield a design with an even higher glide ratio. Structural improvements could also be made. The wings bent up a lot which increased the airplane stability, but also decreased the vertical lift and produced a lot of bending stress in the wings.

The coding project produced a method for visualizing fluid velocity and vorticity near windcraft turbines with the potential for the method to be easily adapted to other simulations run using FLOWUnsteady. A simple guide was produced that details the coding process and the method for creating velocity profiles and vorticity isosurfaces. The coding method produced accurate results, but the method could be improved to run faster as the process detailed in section 3.1 can take a long time to run for many particles.

Appendix A

Aircraft Videos

Glide Test Videos

Glide Test 1

Glide Test 2

Flight Videos

Flight 1

Flight 2

Appendix B

Visualization Code

View the code and simple animations on GitHub here.

```
# Isosurface Creation
# Author: Greg Devenport
# Date: April 20, 2021
# This code creates the files necessary for isosurface creation and velocity
    visualization in ParaView.

using GeometricTools
using LinearAlgebra
using FLOWVPM

vpm = FLOWVPM;
gt = GeometricTools;
UJ = vpm.UJ_direct;

# Use create_iso_stationary for stationary case.
# Use create_iso_circular for curved path case.
# Straight path case to be added in the future if needed.

"""
    create_iso_stationary(file_start, file_end, freestream, data_path,
       pfield_file_name, save_path, vtk_save_name, verbose, center,
       dimensions, divisions)
    For use when vehicle is stationary (turbines, propeller in hover, etc.)
    Inputs are

    'file_start' an int representing the file number start.
    'file_end' an int representing the file number end.
```

```
25      'freestream' which is a vector containing the freestream components used
            in the simulation,
26      'data_path' which is a string where the h5 files are contained,
27      'pfield_file_name' a string specifying the pfield name, usually sim_pfield
28      'save_path' which is a string where you want to store the pfield and/or
            vtk files,
29      'vtk_save_name' which is a string of the vtk file save names,
30      'verbose' which is a bool, setting to true will cause many lines of text
            to be printed as you monitor the code progress.
31      'center' a vector of the origin of the fluid domain.
32      'dimensions' a vector specifying the x,y,z dimensions of the fluid domain.
33      'divisions' a vector specifying the number of divisions in the fluid
            domain in the x,y,z planes.
34  """
35  function create_iso_stationary(;
36      file_start=file_start,
37      file_end=file_end,
38      freestream=freestream,
39      data_path=data_path,
40      pfield_file_name=pfield_file_name,
41      save_path=save_path,
42      vtk_save_name=vtk_save_name,
43      verbose=verbose,
44      center=center,
45      dimensions=dimensions,
46      divisions=divisions,
47      )
48
49      @time begin
50          #--------------------------------------------------------------------
                Extract initial parameters
                ---------------------------------------------------
51          # Extract the length of each side of the fluid domain.
52          x_length = dimensions[1];
53          y_length = dimensions[2];
54          z_length = dimensions[3];
55
56          for i in file_start:file_end
57              # ------------------------------------------------------------
                    Create Fluid Domain
                    ------------------------------------------------------
58
59              # Define the two sets of coordinates needed to define the fluid
                    domain.
60              x1 = center[1] - x_length/2;
61              x2 = center[1] + x_length/2;
62              y1 = center[2] - y_length/2;
63              y2 = center[2] + y_length/2;
64              z1 = center[3] + z_length/2;
65              z2 = center[3] - z_length/2;
```

```
66
67              # Create fluid domain grid. 'divisions' defines the number of
                    nodes in the grid. Number of nodes is (divisions[1]+1)*(
                    divisions[2]+1)*(divisions[3])
68              fdom = gt.Grid([min(x1,x2),min(y1,y2),min(z1,z2)],[max(x1,x2),max(
                    y1,y2),max(z1,z2)],convert(Array{Int64,1}, divisions))
69
70              if verbose println("Creating Isosurface for file $i") end
71
72
73              # Read in the pfield data from the h5 file.
74              X, Gamma, Sigma , lengthX = readh5("$pfield_file_name.$i.h5",
                    data_path);
75
76              if verbose println("Building particle field...number of particles:
                     $lengthX") end
77
78              # ----------------------------------------------------Create
                    Particle Field
                    ------------------------------------------------------
79              # Initialize both particle fields.
80
81              # Pfield from h5 file is recreated here, it will be just as it was
                     in the simulation.
82              pfield_from_h5_file = vpm.ParticleField(lengthX);
83
84              # Pfield with test particles corresponding to the nodes of the
                    fluid domain.
85              pfield_for_fluid_domain = vpm.ParticleField(fdom.nnodes + 1);
86
87              # Add probes to the particle field at the nodes of the grid. Use
                    small values of gamma and sigma.
88              for i in 1:fdom.nnodes
89                  Xprobe = gt.get_node(fdom, i)
90                  vpm.add_particle(pfield_for_fluid_domain, Xprobe, 1e-10*ones
                        (3), 0.01)
91              end
92
93              # Recreate the pfield from the simulation.
94              # It should be noted that the [x,y,z] data are arranged with x,y,z
                     as rows and the various particles as columns.
95              # The same is true of the Gamma data.
96              for i in 1:lengthX
97                  vpm.add_particle(pfield_from_h5_file, X[:,i], Gamma[:,i],
                        Sigma[i])
98              end
99
100             # ---------------------------------------------------------
                    Calculate Vorticity and Velocity
                    ----------------------------------------
```

```
101             if verbose println("Calculating vorticity and velocity..."); println("\t Resetting particle field...") end
102
103             # The pfields must be reset each iteration so that the velocities do not continue to add on top of eachother.
104             vpm._reset_particles(pfield_for_fluid_domain)
105             vpm._reset_particles(pfield_from_h5_file)
106
107             if verbose println("\t Calculating particle on particle interations...") end
108
109             # Calculate the particle on particle interations.
110             UJ(pfield_from_h5_file,pfield_for_fluid_domain)
111
112             if verbose println("\t Calculating velocity...") end
113
114             # Extract the velocity at each node on the fluid grid.
115             Us = [vpm.get_U(P)+freestream for P in vpm.iterate(pfield_for_fluid_domain)]
116
117             if verbose println("\t Calculating vorticity...\n") end
118
119             # Extract the vorticity at each node on the fluid grid.
120             Ws = [vpm.get_W(P) for P in vpm.iterate(pfield_for_fluid_domain)]
121
122             # ------------------------------------------------------Add Solutions to Fluid Domain
                -------------------------------------------
123             # Add the velocity (Us) and vorticity (Ws) data to the fluid domain.
124             gt.add_field(fdom, "U", "vector", Us, "node")
125             gt.add_field(fdom, "W", "vector", Ws, "node")
126
127             # Generate the file number to match the input h5 file. Output in .%4d format (0001, 0010, 0100, 1000).
128             if i < 10
129                 file_number = "000$i";
130             elseif i < 100
131                 file_number = "00$i";
132             elseif i < 1000
133                 file_number = "0$i";
134             else
135                 file_number = "$i";
136             end
137
138             # ------------------------------------------------------Save VTK Files
                -------------------------------------------------------------
139             # Save the grid as a VTK file.
140             gt.save(fdom,"$save_path$vtk_save_name";num=i)
```

```
141
142         end
143     end
144 end
145
146
147 """
148     create_iso_circular(file_start, file_end, freestream, data_path,
        pfield_file_name, save_path, vtk_save_name, verbose,
149     circular, center, dimensions, v_vehicle, divisions, t_total,
        rotation_center)
150     For use when vehicle moves in circular path (windcraft, etc.)
151     Inputs are
152     'file_start' an int representing the file number start.
153     'file_end' an int representing the file number end.
154     'freestream' which is a vector containing the freestream components used
        in the simulation,
155     'data_path' which is a string where the h5 files are contained,
156     'pfield_file_name' a string specifying the pfield name, usually "
        sim_pfield"
157     'save_path' which is a string where you want to store the pfield and/or
        vtk files,
158     'vtk_save_name' which is a string of the vtk file save names,
159     'verbose' which is a bool, setting to true will cause many lines of text
        to be printed as you monitor the code progress.
160     'circular' a bool set to true if the simulation involves a circular path.
161     'center' a vector of the origin of the fluid domain.
162     'dimensions' a vector specifying the x,y,z dimensions of the fluid domain.
163     'v_vehicle' is the velocity of the vehicle.
164     'divisions' a vector specifying the number of divisions in the fluid
        domain in the x,y,z planes.
165     't_total' the total time the simulation ran for.
166     'rotation_center' a vector specifying the point around which the vehicle
        moves about.
167 """
168 function create_iso_circular(;
169     file_start=file_start,
170     file_end=file_end,
171     freestream=freestream,
172     data_path=data_path,
173     pfield_file_name=pfield_file_name,
174     save_path=save_path,
175     vtk_save_name=vtk_save_name,
176     verbose=verbose,
177     center=center,
178     dimensions=dimensions,
179     v_vehicle=v_vehicle,
180     divisions=divisions,
181     t_total=t_total,
182     rotation_center=rotation_center
```

```
183     )
184     #----------------------------------------------------------Initial
            calculations for circular path-------------------------------------
185     #####
186     # All circular path code is fairly new and may have issues. Initial
            results seem correct however.
187     #####
188     @time begin
189         # Extract the length of each side of the fluid domain.
190         x_length = dimensions[1];
191         y_length = dimensions[2];
192         z_length = dimensions[3];
193
194         # Change this if circular path is in a different plane. This is
                currently set for y/z plane.
195         r = sqrt((center[2]-rotation_center[2])^2 + (center[3]-rotation_center
                [3])^2);
196
197         # Calculate the total number of steps, most likely the same as
                file_end.
198         n_steps = file_end - file_start;
199
200
201         for i in file_start:file_end
202
203             #-----------------------------------------------------------Define
                    parameters for circular path
                    ---------------------------------------
204             # This is the real time used to ensure the circular path of the
                    fluid domain matches the circular path of the vehicle in the
                    simulation.
205             t = i*(t_total/n_steps)
206
207             # So far this only works for a circular path in a plane (x/y, x/z,
                     y/z) and not in three dimensions.
208             # Change z1, z2, y1, y2 to the appropriate variables so the
                    circular path is in the desired plane.
209             z1 = rotation_center[3] + r*cos(t*v_vehicle/r) - z_length/2 +
                    center[3];
210             z2 = rotation_center[3] + r*cos(t*v_vehicle/r) + z_length/2 +
                    center[3];
211             y1 = rotation_center[2] + r*sin(t*v_vehicle/r) + y_length/2 +
                    center[2];
212             y2 = rotation_center[2] + r*sin(t*v_vehicle/r) - y_length/2 +
                    center[2];
213
214             # These are constant during the circular path.
215             x1 = rotation_center[1] - x_length/2 + center[1];
216             x2 = rotation_center[1] + x_length/2 + center[1];
217
```

```
218             # Define the bounds of the fluid domain.
219             circle_path_coordinates = [[min(x1,x2),min(y1,y2),min(z1,z2)],[max
                    (x1,x2),max(y1,y2),max(z1,z2)]]
220
221             # ----------------------------------------------------------Create
                    Fluid Domain
                    ----------------------------------------------------------
222             # Create fluid domain grid. 'divisions' defines the number of
                    nodes in the grid. Number of nodes is (divisions[1]+1)*(
                    divisions[2]+1)*(divisions[3])
223             fdom = gt.Grid(circle_path_coordinates[1],circle_path_coordinates
                    [2],convert(Array{Int64,1}, divisions))
224
225             # Print file number code is running on.
226             if verbose println("Creating Isosurface for file $i") end
227
228             X, Gamma, Sigma , lengthX = readh5("$pfield_file_name.$i.h5",
                    data_path);
229
230             if verbose println("Building particle field...number of particles:
                     $lengthX") end
231
232             # ----------------------------------------------------------Create
                    Particle Field
                    ----------------------------------------------------------
233             # Initialize both particle fields.
234
235             # Pfield from h5 file is recreated here, it will be just as it was
                     in the simulation.
236             pfield_from_h5_file = vpm.ParticleField(lengthX);
237
238             # Pfield with test particles corresponding to the nodes of the
                    fluid domain.
239             pfield_for_fluid_domain = vpm.ParticleField(fdom.nnodes + 1);
240
241             # Add probes to the particle field at the nodes of the grid. Use
                    small values of gamma and sigma.
242             for i in 1:fdom.nnodes
243                 Xprobe = gt.get_node(fdom, i)
244                 vpm.add_particle(pfield_for_fluid_domain, Xprobe, 1e-10*ones
                        (3), 0.01)
245             end
246
247             # Recreate the pfield from the simulation.
248             # It should be noted that the [x,y,z] data are arranged with x,y,z
                     as rows and the various particles as columns.
249             # The same is true of the Gamma data.
250             for i in 1:lengthX
251                 vpm.add_particle(pfield_from_h5_file, X[:,i], Gamma[:,i],
                        Sigma[i])
```

```
252             end
253
254             # ----------------------------------------------------------
                    Calculate Vorticity and Velocity
                    ----------------------------------------
255             if verbose println("Calculating vorticity and velocity...");
                    println("\t Resetting particle field...") end
256
257             # The pfields must be reset each iteration so that the velocities
                    do not continue to add on top of eachother.
258             vpm._reset_particles(pfield_for_fluid_domain)
259             vpm._reset_particles(pfield_from_h5_file)
260
261             if verbose println("\t Calculating particle on particle
                    interations...") end
262
263             # Calculate the particle on particle interations.
264             UJ(pfield_from_h5_file,pfield_for_fluid_domain)
265
266             if verbose println("\t Calculating velocity...") end
267
268             # Extract the velocity at each node on the fluid grid.
269             Us = [vpm.get_U(P)+freestream for P in vpm.iterate(
                    pfield_for_fluid_domain)]
270
271             if verbose println("\t Calculating vorticity...\n") end
272
273             # Extract the vorticity at each node on the fluid grid.
274             Ws = [vpm.get_W(P) for P in vpm.iterate(pfield_for_fluid_domain)]
275
276             # ----------------------------------------------------------Add
                    Solutions to Fluid Domain
                    ---------------------------------------------
277             # Add the velocity (Us) and vorticity (Ws) data to the fluid
                    domain.
278             gt.add_field(fdom, "U", "vector", Us, "node")
279             gt.add_field(fdom, "W", "vector", Ws, "node")
280
281             # Generate the file number to match the input h5 file. Output in
                    .%4d format (0001, 0010, 0100, 1000).
282             if i < 10
283                 file_number = "000$i";
284             elseif i < 100
285                 file_number = "00$i";
286             elseif i < 1000
287                 file_number = "0$i";
288             else
289                 file_number = "$i";
290             end
291
```

```
292             # ----------------------------------------------------------Save VTK
                    Files
                  ------------------------------------------------------------
293             # Save the grid as a VTK file.
294             gt.save(fdom,"$save_path$vtk_save_name";num=i)
295         end
296     end
297 end
```

Appendix C

Isosurface Creation Guide

Vorticity and Velocity Visualization

Greg Devenport

April 22, 2021

1 Vorticity Isosurface and Velocity Profile Creation

Vorticity isosurfaces and velocity profiles from FLOW Unsteady simulations can be visualized in ParaView after extracting the needed data from the simulation particle field. Code that does this can be found here. The basic steps the code takes are as follows. (For a more detailed description of the steps, see code documentation)

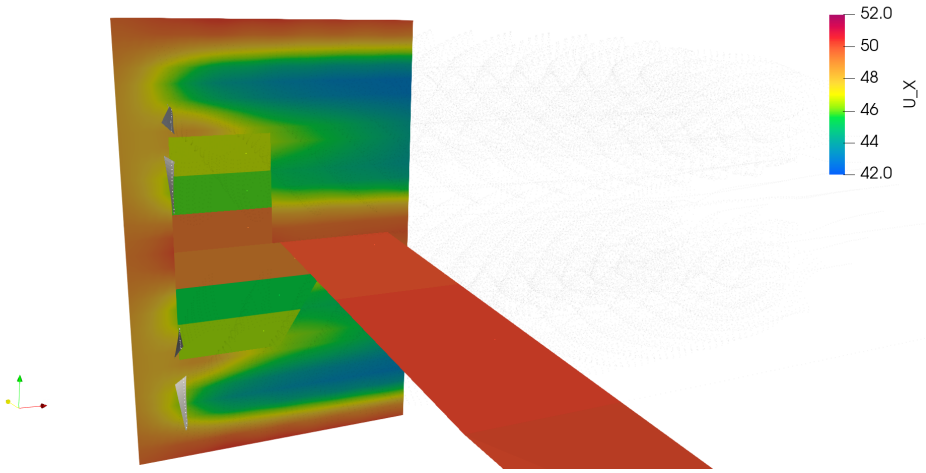
- For a given time step, read in the needed data (position, gamma, sigma) from the simulation particle field h5 file.
- Using this data, recreate the particle field as would be found in the simulation.
- Create a fluid domain grid to which velocity and vorticity solutions will be added later.
- Create a particle field of test particles to be used as probing points. Use small values of gamma and sigma so the particles do not affect the velocity and vorticity solutions. This test particle field will have the same number of particles as nodes in the fluid domain grid and each particle corresponds to a node in the fluid domain grid.
- Calculate how the simulation particle field interacts with the test particle field.
- Probe each particle in the test particle field and extract the velocity and vorticity.
- Add the calculated velocity and vorticity vectors to the corresponding fluid domain grid node.
- Export the fluid domain with the velocity and vorticity solutions as a vtk file.
- Repeat the above process for the desired number of time steps.

2 ParaView

- In ParaView, open the vtk files produced by the code, as well as any other files for visualization (wings, rotors, etc.).

2.1 Velocity Profile

- To view the velocity, change the coloring of the fluid domain from "Solid Color" to "U".
- If desired extract a specific component from the fluid domain vtk file by applying the calculator filter. (mag(U), U_X, etc.)
- A slice filter can be applied to the fluid domain or calculator filter to achieve the result as shown in figure 1. The velocity profile is shown with the wing and pylon showing the velocity calculated in the FLOW Unsteady simulation.



- Isosurfaces require scalar values. Use the calculator filter to extract scalar values from the fluid domain (mag(W), W_X, etc.).
- Apply a contour filter to the calculator filter and set the desired value range for the isosurfaces (an example is shown in figure 2).
- The above process can also be used to create velocity isosurfaces.

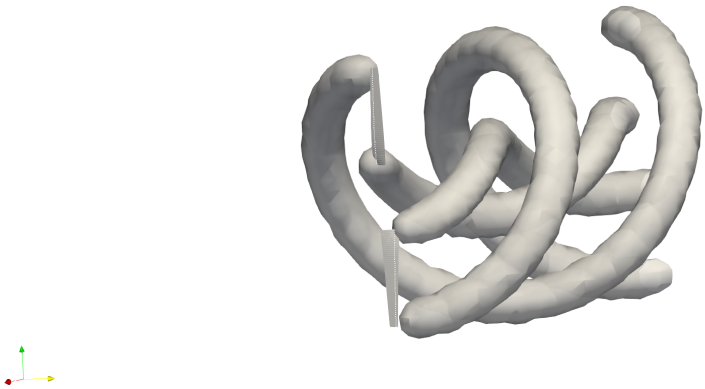


Figure 2: Vorticity isosurfaces trailing from turbine blades

2.3 Curved Path Velocity Slice

For the case of a windcraft traveling in a circular path, the fluid domain grid needs to travel with the windcraft. With a few more inputs than the stationary case, the above code can achieve this. Once the vtk files have been created, perform the following steps for the slice to move with constant orientation with respect to the vehicle. These steps may need to be slightly modified depending on the simulation orientation.

- Perform the same steps as in 2.1, to create the desired velocity slice for the starting time.
- Go to the last desired time step. The slice will have moved and its orientation with respect to the vehicle will be different.
- Change the "Origin" and "Normal" parameters in the slice filter until the slice again has the desired orientation.
- As shown in figure 3, use the drop down next to the blue plus sign to select the name of the slice. In the drop down to the right of that, select "Normal(1)". Click the plus sign to add this.
- Double click anywhere on "Slice1 - Slice Type - Normal(1)" to adjust the values at the beginning and end of the animation sequence. The end Normal(1) value should be changed to the value just used to adjust the slice orientation, and the beginning Normal(1) value should be changed to the value used to originally orient the slice. The rotation of Normal(1) will be interpolated using the starting and ending values provided.
- Repeat this process with other "Normal" and "Origin" components as needed.
- The slice will now transform in time and space as the simulation vehicle moves.
- Depending on how much the vehicle moves, multiple interpolation points may need to be added between the first and last time step to ensure the transformation happens correctly.

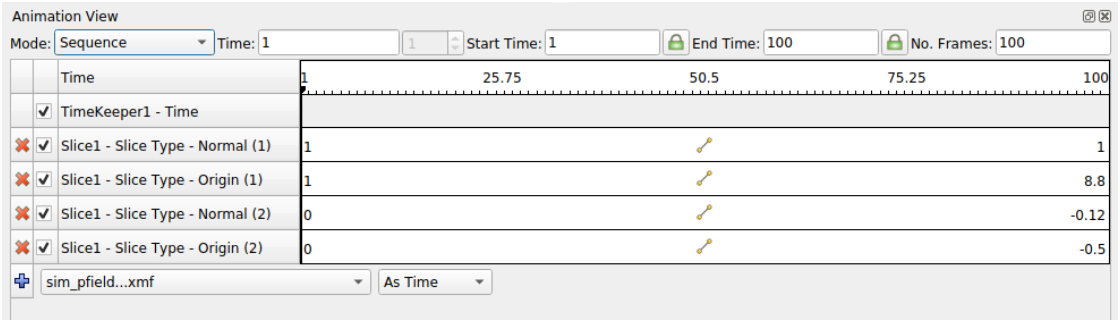


Figure 3: Animation view in ParaView

References

[1] Eduardo Alvarez, FLOWUnsteady, https://github.com/byuflowlab/FLOWUnsteady

[2] X Development, "Makani", https://x.company/projects/makani/

[3] A. Ning, "Flight Vehicle Design", (2018), Books, 26, https://scholarsarchive.byu.edu/books/26

[4] E. Alvarez and A. Ning, "Development of a Vortex Particle Code for the Modeling of Wake Interaction in Distributed Propulsion", (2018), p. 2

[5] J. Mehr, E. Alvarez, A. Ning, "Unsteady Aerodynamic Analysis of Wind Harvesting Aircraft", (2020), Faculty Publications, 4054, https://scholarsarchive.byu.edu/facpub/4054