12-20-2020

Validating an Open-Source UVLM Solver for Analyzing Flapping Wing Flight

Cameron Urban
Washington University in St. Louis

E37 400-09: Independent Study

Fall 2020

Validating an Open-Source UVLM Solver for Analyzing Flapping Wing Flight

By: Cameron Urban

Research Advisor: Dr. Ramesh Agarwal

Department of Mechanical Engineering and Materials Science

Washington University in St. Louis

December 20, 2020

Table of Contents

Figures

Abstract

Scientists have long used the unsteady vortex lattice method (UVLM) to simulate flapping wing flight. However, until recently, there has not been an available open-source UVLM solver designed explicitly for this field. Ptera Software is the only open-source UVLM solver that can simulate flapping wing aerodynamics without the modifications required by other open-source solvers. This report documents the next step in the software's development: validation of its results. Comparing Ptera Software's output to high-fidelity experimental data of the pressures on a flapping wing robot shows that the simulated results predict the trends and magnitudes of the net lift over time with good accuracy. The present results demonstrate that Ptera Software correctly implements the UVLM and can simulate flapping wing flight with reasonable accuracy under this method's assumptions.

Introduction

Animals are better at flying than any human invention. While animals may not be able to fly as fast as a jetliner or carry as much weight as a cargo plane, they fly with higher maneuverability, adaptability, and grace than anything aerospace engineers have invented before or after the Wright brothers took off from Kitty Hawk. In the words of aerodynamicists McMasters and Henderson, "…humans fly commercially or recreationally, but animals fly professionally" (Shyy et al., 2007).

These advantages indicate that natural flight is a rich area for research that could improve human flying machines. Unfortunately, analyzing how animals fly is extremely difficult. When I first became interested in natural flight, I assumed that the simplistic methods used to analyze conventional aircraft would be applicable. Instead, I realized that the high performance of flying animals is matched by the complexity of the aerodynamic mechanisms they utilize. Thankfully, I could begin my research on the foundations laid by other aerodynamicists who have worked tirelessly to unlock the secrets of flapping wing flight.

One of the tools discovered by these researchers is the unsteady vortex lattice method (UVLM). This tool, described in more mathematical detail in the literature (Katz & Plotkin, 2001) than I will discuss here, is a well-established algorithm for analyzing unsteady aerodynamics. It has also been used before to simulate flapping wings. As a final advantage over other methods, the UVLM also strikes a favorable balance between accuracy and the computational time required to run the simulations. For these reasons, I selected it as my tool of choice in exploring natural flight.

Unfortunately, I quickly stumbled into a roadblock. While many papers have been written using the UVLM to analyze flapping wings, I could not find any commercial or open-source UVLM programs capable of doing so without significant modifications to their source code. Therefore, I decided to write my own.

After a few months of work, I released Ptera Software, the only open-source UVLM designed to analyze flapping wings. It is free to use, and I hope that it will inspire further research into the field of animal aerodynamics and continue to improve as a community of scientists uses and modifies it. For more information on Ptera Software, or to download the source code, visit the project page on GitHub.

This paper documents the next step in Ptera Software's development: validating its results against experimental results. The first step in this process was reviewing the literature to find appropriate experimental data. Next, I used Ptera Software to emulate the experimental setup that produced this data. Finally, I ran Ptera Software's UVLM solver on this emulation and compared my simulated findings against those from the original experiment.

Literature Review

Analyzing Flapping Wings with the UVLM

The literature has demonstrated that the UVLM can accurately predict the aerodynamic forces and moments exerted on flapping wings by comparing their results to data gained experimentally. For example, researchers Fritz and Long validated plunging and pitching UVLM simulations against experimental data and analytical solutions (Fritz & Long, 2004). These experiments have been corroborated by Lambert and Dimitriadis, who found that the UVLM accurately modeled the forces on a flapping micro air vehicle (MAV) as long as the flow around its wings remained attached (Lambert & Dimitriadis, 2014).

The UVLM has also been used to simulate the flapping wing flight of real animals. For example, Gardiner et al. used the UVLM to study the flight of barnacle geese (Gardiner et al., 2013). However, the researchers did not provide experimental data that validated their simulated results. Some authors have even used a modified form of the UVLM to simulate the flight of insects (Nguyen et al., 2016).

Almost every paper in this field uses a home-grown code developed by each researcher's laboratory. There is no widely accepted, easily accessible framework for conducting low-fidelity flapping wing research. Therefore, once validated, Ptera Software could aggregate each institution's disparate advances and eliminate the barrier to entry of developing a solver for those newly interested in this field.

Experimental Data for Validation

To validate the theoretical results produced by Ptera Software, I needed experimental data to use as a benchmark. I found this data in "Experimental and Analytical Pressure Characterization of a Rigid Flapping Wing for Ornithopter Development" (Yeo et al., 2011). This paper details the

analysis of a robotic ornithopter, an umbrella term for any type of flapping wing aircraft. Yeo et al. mounted their ornithopter to a test stand within a wind tunnel, as shown in Figure 1.

While flapping in the wind tunnel, the authors gathered aerodynamic data from the robot via an array of nine differential pressure sensors mounted on one of the wing halves. The sensors measured the difference in pressure between the wing's lower and upper surfaces.

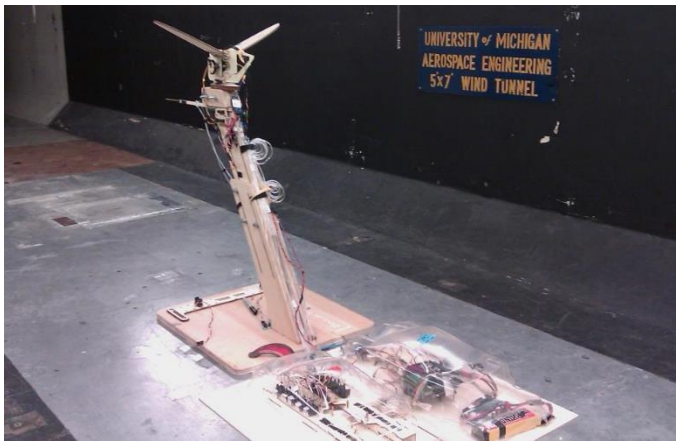

Figure 1: Robotic Flapping Wing Robot in Wind Tunnel (Yeo et al., 2011)

I chose the data from their paper to validate Ptera Software for three reasons. (1) The data provided by this paper is well suited for comparison with the data from the UVLM. By using pressure sensors instead of a more traditional force sensor, Yeo et al. bypassed the experimental confounding factor of separating aerodynamic and inertial loads. (2) Yeo et al. analyzed forward flapping wing flight along with hovering flapping wing flight. Forward flapping wing flight data is critical because the traditional UVLM is not the proper tool for analyzing hovering flapping wing flight due to the prevalence of separated flow under hovering conditions. Flow separation is a viscous phenomenon where a fluid ceases to follow the curvature of a surface. A real-world example of this is an aircraft stalling at too steep an angle of attack for its airspeed. As the UVLM is an inviscid solver, it cannot account for separation. However, flow separation is significantly less prevalent during forward flapping wing flight because the effective angle of attack is reduced. Therefore, I use the forward flapping wing flight case for my validation. (3) The University of Michigan, where this research took place, is a well-respected institution for the study of aerodynamics, which lends credence to the quality of the data collected.

Methodology

After I selected the experimental data to use for the validation, it needed to be processed, and the geometry of Yeo et al.'s robotic ornithopter modeled in Ptera Software. Additionally, I had to

check the results for convergence with respect to temporal and spatial discretization. The steps of this procedure are shown in Figure 2.

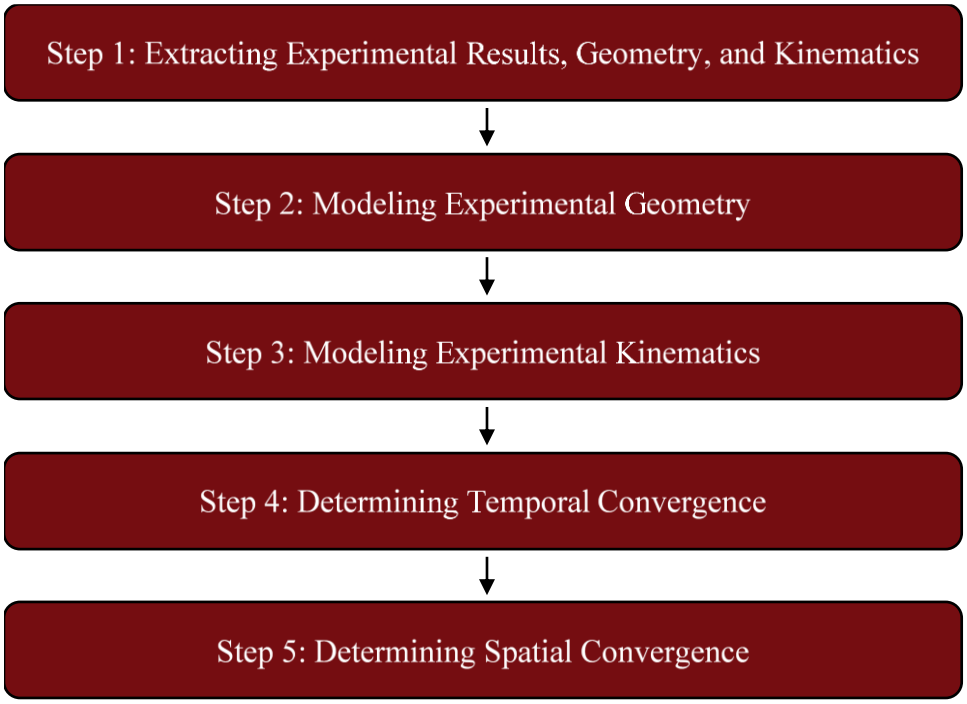

Figure 2: The process of running the validation study

Extracting Experimental Results, Geometry, and Kinematics

I chose to validate against the experiment referred to in Yeo et al., 2011 as case C. The data from this run is shown in Figure 3, reprinted from Yeo et al., 2011, which contains five subplots. The upper left subplot is a sketch of the nine pressure sensors' approximate positions on the instrumented wing. The lower left subplot shows Yeo et al.'s measured versus predicted wing flap angle during the test. Each of the right three subplots displays the cycle-averaged pressure data for three of the nine sensors. The upper right, middle right, and lower right subplots show the data for the "blue," "orange," and "green" sensors, respectively. The upper left subplot can be used to identify each sensor's color and number designations.

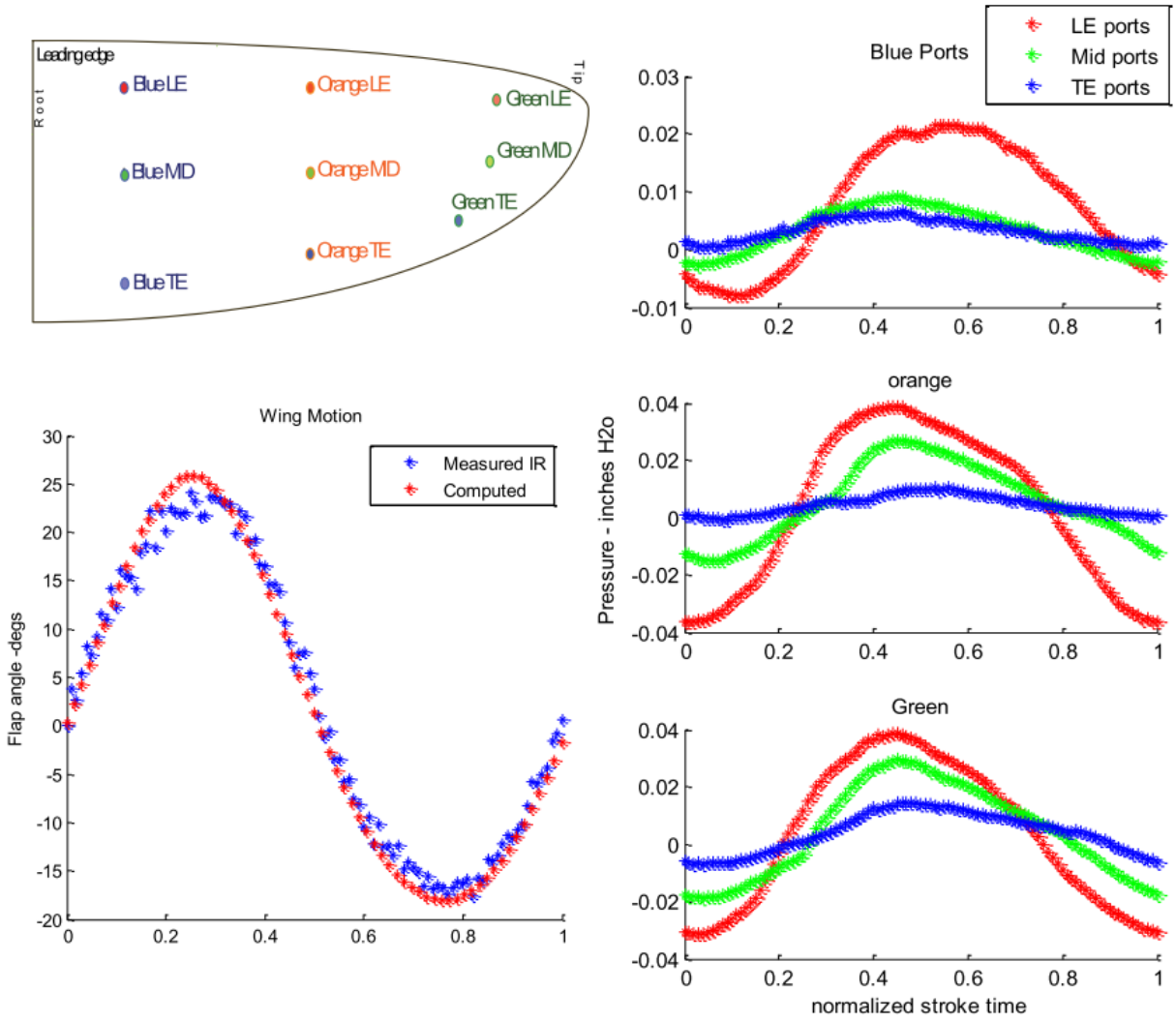

Figure 3: Data from Flapping Wing Robot at Flight Condition of Interest (Yeo et al., 2011)

Since Yeo et al. did not provide the raw data in tabulated form, I used the popular data analysis tool WebPlotDigitizer to extract approximated raw pressure data from screenshots of Figure 3's pressure plots that I uploaded to the tool's website (Rohatgi, 2020). After appropriately scaling the axes, the program automatically detected the positions of the graph's data points, and compiled them into a downloadable comma-separated values (CSV) file.

Modeling Experimental Geometry

The second step was to model the geometry of Yeo et al.'s robotic wings in Ptera Software. I first saved a plot from Yeo et al., 2011 of the wing's shape and sensor locations, reprinted here as

Figure 4. Then, I uploaded the plot to WebPlotDigitizer, which provided interpolated coordinates of the wing planform's curve, shown in Figure 4 as a solid black line.

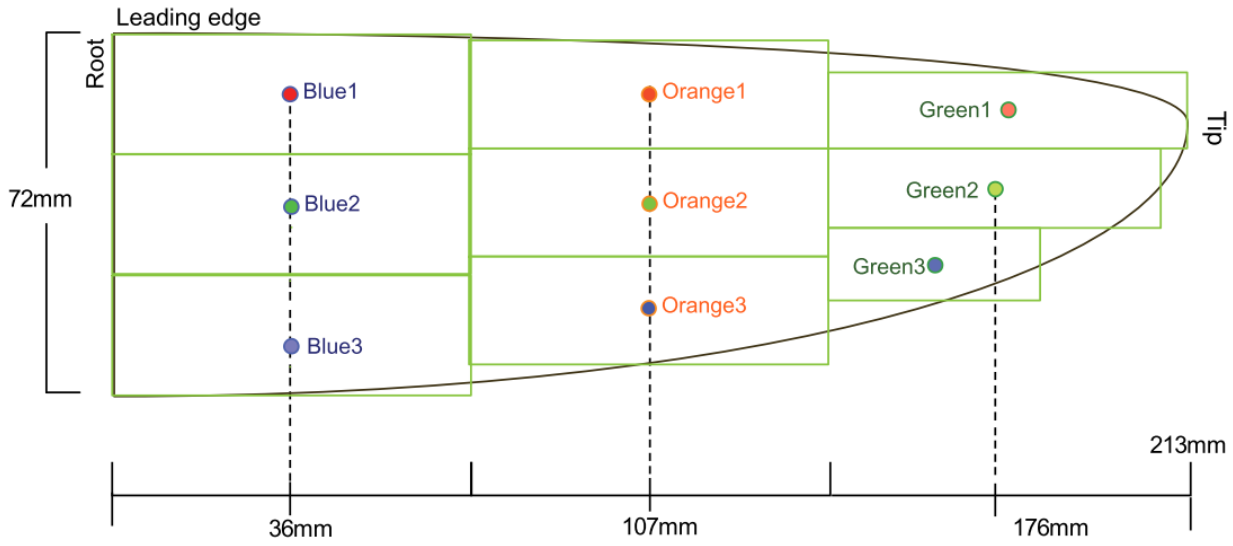

Figure 4: The geometry of the robotic half wing (Yeo et al., 2011)

I was then able to discretize the wing in the spanwise direction into any number of uniformly-spaced wing cross sections. As shown in Figure 4, the experimental wing planform extends to a smooth tip. I did not design Ptera Software to analyze wings with a tip chord length of zero. So, to avoid the possibility of numerical errors, I sliced off the outermost five millimeters of the wingtip so that the last wing cross section had a non-zero chord length. For more details on this process, see the code in Appendix A.

From Figure 4, I also extracted the approximate areas upon which each pressure acts. The figure represents these areas as the green rectangles around each of the ports. Using this data, the normal force on each rectangle is simply the pressure at its port multiplied by its area. The lift on each panel is the vertical component of this force at that particular time step, which I calculated by multiplying the normal force by the cosine of the current flapping angle. Finally, the total lift on the robot is the sum of the lift forces on each rectangle multiplied by two (because there are two wing halves, and only one has pressure sensors).

While necessary, the pressure data and the geometry dimension extraction step introduces potential errors into my validation because the results are approximate. Future work would allow for collaboration with Yeo et al. to rerun this validation study with the exact experimental data.

Modeling Experimental Kinematics

The third step is to model the experimental kinematics. In a separate paper, the authors of "Experimental and Analytical Pressure Characterization of a Rigid Flapping Wing for Ornithopter Development" analyzed the same robot's flapping mechanism to characterize the flapping angle as a function of time (Yeo et al., 2012). They found that the flapping was experimentally repeatable and well modeled by a fourth-order Fourier series. Figure 5 shows the Fourier series' coefficients, and the lower left subplot in Figure 3 displays a comparison between the experimental and modeled kinematics of the flapping.

Coefficient	Value
a0	0.0354
a1	4.10E-05
b1	0.3793
a2	-0.0322
b2	-1.95E-06

Coefficient	Value
a3	-8.90E-07
b3	-0.0035
a4	0.00046
b4	-3.60E-06
w	6.231

Figure 5: The Fourier series coefficients for the flapping angle equation (Yeo et al., 2012)

Using these Fourier series coefficients and the flapping frequency of the test I was emulating, I wrote a Python function that Ptera Software used to simulate the flapping kinematics.

Determining Temporal Convergence

The UVLM is a time-stepping simulation, and the fourth step in setting up the study is picking how much time to simulate. During each time step, the wing sheds a new row of ring vortices into its wake. The panels on the wing experience the effect of the panels in the wake during the next time step. If the simulation were allowed to run for an infinitely long time, there would be an infinitely long sheet of wake panels stretching behind the wing. At this point, the pressure vs. time graphs for each subsequent flapping cycle would be identical. We would say that the system has reached a quasi-steady state (quasi because the pressure distributions are still varying over each flap cycle, but this variation is constant from one flap cycle to another).

However, simulating for an infinitely long time is impractical. Thankfully, as the wake ring vortices get further away from the wing, their effects decrease rapidly. Therefore, there should be some number of flap cycles, after which additional flap cycles produce only a negligible difference in the pressure vs. time functions.

I found this threshold to be three flap cycles. I determined this after running several test simulations for one, two, three, and four flap cycles with different spatial discretizations and observing that there was qualitatively no difference in the results for two and three cycles.

Determining Spatial Convergence

The final step in running the experiment in Ptera Software is to determine how to arrange the wing panels. Ptera Software's implementation of the UVLM first turns a wing into a 2D surface and then discretizes it into an array of rectilinear panels. Generally, the more panels used, the more accurate the result. Additionally, I weighed several other considerations before choosing the number of panels and determining their shape.

- Spacing: In many steady vortex solvers, the panels are smaller and more tightly grouped near the wing's leading edge, trailing edge, root, and tip. Researchers use this spacing scheme because the pressure gradient is usually higher at these locations. However, most unsteady solvers use a uniform spacing, at least in the chordwise direction, so that the wake panels are roughly the same area as wing panels that shed them. This choice satisfies the wake transport equation, one of the governing laws of the UVLM (Binder et al., 2017). While not required by the wake transport equation, I used uniform spacing for the spanwise panels to reduce the code required to run these cases. Future work should revisit this decision.
- Aspect Ratio: The panel's aspect ratio is its average chordwise dimension divided by its average spanwise dimension. This value should be kept near one to reduce computational error (*Guidelines for QFLR5 v0.03 XFLR5 Analysis of Foils and Wings Operating at Low Reynolds Numbers*, 2009).
- Computation Time: The computation time increases dramatically with the number of panels, specifically with the number of chordwise panels. Therefore, I used the smallest number of panels required to achieve convergence.

Based on the above principles, I decided to space my panels uniformly, keep their aspect ratios as close to one as possible, and design a small study increasing the number of chordwise panels from a bare minimum amount until I saw qualitative convergence. I saw this convergence after reaching five chordwise panels with 18 spanwise panels.

Results

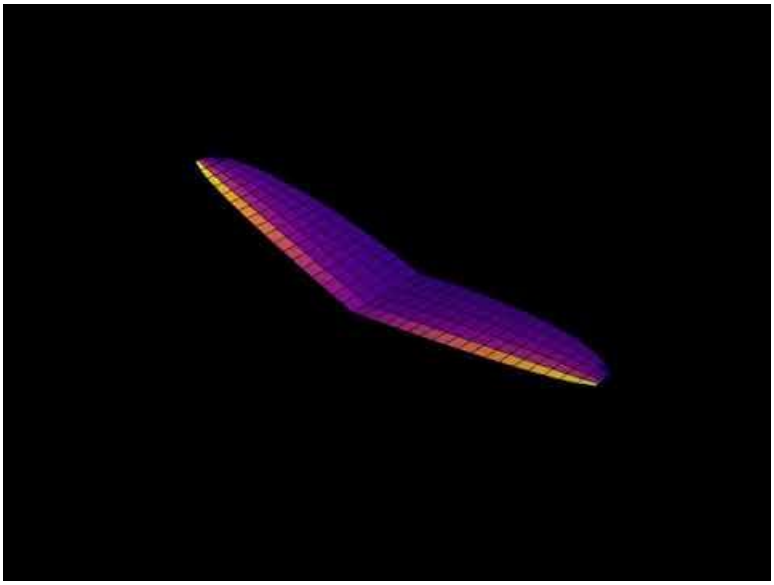

Figure 6: A close-up, slow-motion animation of the simulated experiment's wings. The wing panel colors represent the magnitudes of the pressures. Hot colors indicate higher pressures on the lower surfaces and vice versa. Click to launch the video.

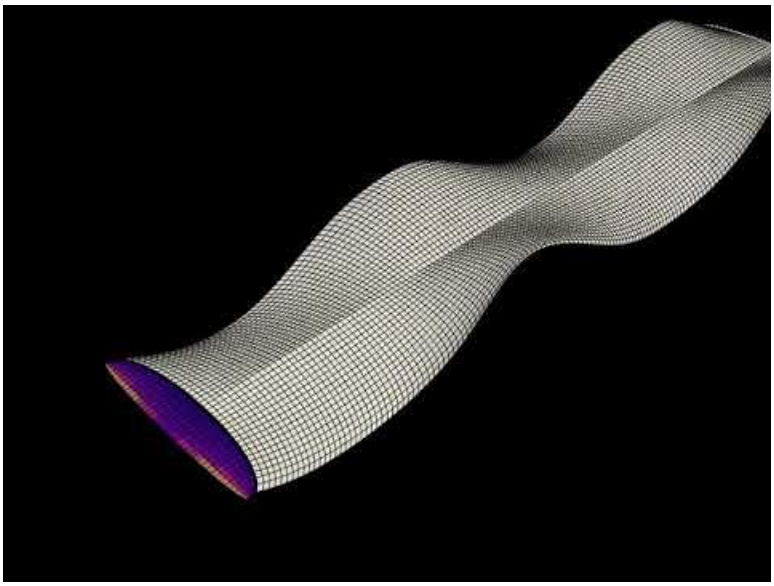

Figure 7: A slow-motion animation of the simulated experiment's wings and wake. The wing panel colors represent the magnitudes of the pressures. Hot colors indicate higher pressures on the lower surfaces and vice versa. The white vortex rings represent the wake shed from the trailing edge. Click to launch the video.

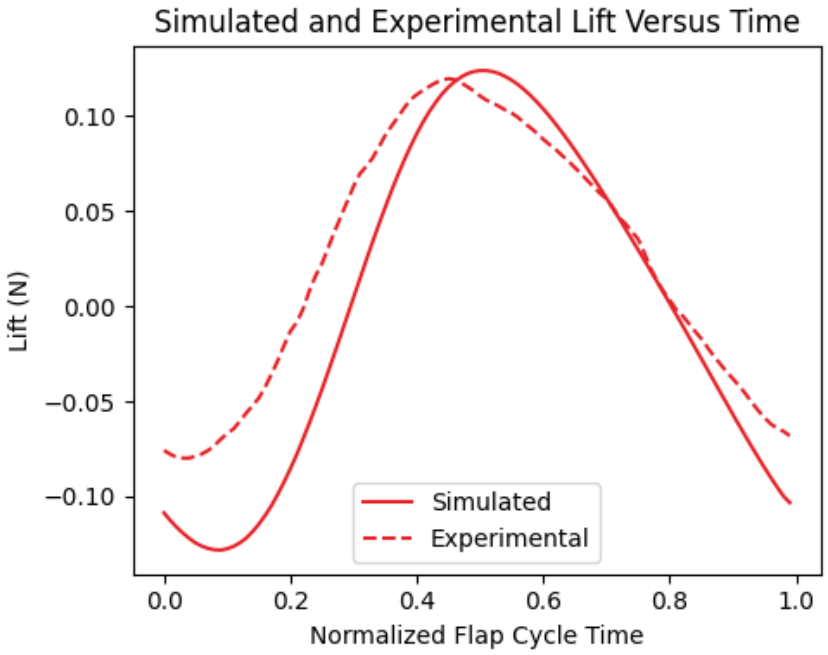

Figure 8: The net simulated and experimental lift across the entire wing over a flap cycle normalized from zero to one. The simulated lift values comprise those calculated during the final flap.

Discussion

Simulation Computational Time

The simulation of the fully converged configuration took just under 22.5 minutes to run on my laptop, which has an Intel® Xeon® E3-1505M v5 CPU running at 2.81 GHz base speed and 15.3 GB of usable RAM. This converged configuration used a prescribed wake, five uniformly spaced chordwise panels, 18 uniformly spaced spanwise panels, and simulated three flap cycles. I discretized these cycles into 224 time steps, the difference in time between each being roughly 4.1 milliseconds. On the last time step, Ptera Software was analyzing a total of 4,104 ring vortices. For reference, this translates to multiple calculations involving matrices with hundreds of thousands of elements during each of the later time steps.

While 20 minutes is a reasonable amount of computational time for this simulation type, I plan to significantly reduce it with small changes to Ptera Software's source code. For example, I will modify Ptera Software to detect symmetric geometry and operating conditions, such as the two sides of the flapping wing in this validation study. Once detected, the software would copy the results of any required computations from one of the sides to the other. Additionally, I will modify

Ptera Software to compute the panels' pressures only during the last flap cycle. Finally, I could parallelize large parts of the program to take advantage of CPUs with multiple cores.

These modifications will reduce the computational time to the point where Ptera Software's users could run large scale optimization studies.

Simulation Results

Figure 6 and Figure 7 are embedded links to the videos produced by the simulation. These visuals are useful in determining quickly if something obvious is wrong with the solver. For example, the videos show that the wing is the correct shape and is flapping as expected. Additionally, the pressures on two wing halves look symmetric throughout, as they should be given that the sides of the robot are identical and experienced identical conditions.

Figure 8 shows that the fully converged simulation models the experimental lift trends and magnitudes relatively well. The simulated lift had a mean absolute error (MAE) of 0.0291 N over the flap cycle with respect to the experimental lift. The root mean square (RMS) of the experimental lift values is 0.0717 N.

I chose MAE as an accuracy metric instead of mean absolute percent error (MAPE) because both the experimental and simulated lift trends oscillate around 0 N. A percent error method is ill-equipped to deal with values near zero, so I chose this error-based approach instead.

Inaccuracies

The two noticeable inaccuracies in the simulated lift trend are an overall right phase shift and an overestimation of the lift force magnitude in the first and final quarter of the flap cycle. The phase shift, while a smaller effect than the magnitude differences, is more difficult to explain. Previous papers have discussed how procedures for averaging experimental data over a cycle could affect the observed phase (Lambert et al., 2017). However, without access to the raw experimental data from Yeo et al., it is impossible to investigate this further.

The negative lift magnitude overestimation during the first and last quarter of the flapping cycle is likely caused by flow separation. Flow separation, an aerodynamic phenomenon where the streamlines of a fluid detach from a surface, is created by viscosity. As the UVLM assumes the fluid is inviscid, it cannot predict this behavior. As shown in Figures 6 and 7, the wing starts its cycle halfway through the upstroke. Therefore, the separated flow conditions correspond to the wing's upstroke. During the upstroke, the wing experiences a highly negative effective angle of attack, likely caused the flow on the wings' lower surfaces to separate due to a strong adverse pressure gradient.

Additionally, inconsistencies in Yeo et al., 2011 may have contributed to inaccuracies in my results. In the report, the authors wrote on two separate occasions that the free stream velocity for the case I simulated was 2.9 m/s. However, in a later section, this was changed to 2.8 m/s. I chose to use the more frequently cited value, 2.9 m/s, for my study. Future work should include contacting the original authors to resolve this inconsistency.

The literature documents work to modify the UVLM to correct for flow separation effects. The most promising attempt modified the UVLM to begin shedding vortex rings off the wing's leading edge after a certain effective angle of attack was reached (Roccia et al., 2013). Future work should implement this method in Ptera Software and analyze its results.

Limitations

While this study's results are promising, researchers should use Ptera Software with an understanding of when its assumptions lose validity. For example, the UVLM assumes inviscid, irrotational, and incompressible flow. While no real flow exactly satisfies these requirements, in practice, they imply that Ptera Software will only produce accurate results for objects operating at relatively high Reynolds numbers, with attached flow, and whose airspeed is much lower than the speed of sound.

Additionally, this software is in its early stages of development. It should not be used to make safety-critical design choices. For situations where high reliability is necessary, run preliminary studies in Ptera Software to iteratively approach a workable solution, then use a more well established and higher fidelity simulation to check the result.

Conclusions

Based on the results of this validation study, I conclude that Ptera Software accurately implements the UVLM. I made this conclusion based on good agreement between the trends and magnitudes of Yeo et al.'s experimental data, and Ptera Software's simulated results of the total lift force on a flapping wing robot

The simulated and experimental lift force time histories differed in two noticeable ways. (1) The simulated force plot leads the experimental force plot by a slight phase shift. (2) The simulated results show the wings producing a significantly more negative lift force during the first and last quarter of the flap cycle, which corresponds to the robot's upstroke. The phase shift may be due to errors in how Yeo et al. cyclically averaged their experimental results. To investigate this further would require access to the researchers' raw data. The difference in magnitude is most likely due to flow separation on the wings' bottom surfaces during its upstroke.

Future work should increase the speed of Ptera Software using various computational techniques, such as parallel processing and symmetry analysis. Additionally, a collaboration with the experimental study's original authors could improve this study's results. Such a collaboration would provide access to the actual wing geometry and raw data, eliminating any error associated with my attempts to extract this data from the paper and eliminating errors from the typos in the original document. Finally, Ptera Software should be modified to account for flow separation using a leading-edge vortex scheme described in the literature (Roccia et al., 2013). The inclusion of leading-edge vortices could increase Ptera Software's accuracy for all use cases and expand its capabilities to analyze hovering flapping wing flight.

In the words of statistician George Box, "All models are wrong, but some are useful." In this spirit, researchers should only use Ptera Software within the bounds of the UVLM assumptions. Additionally, more well-established simulation tools or hand calculations must support any safety-critical engineering decisions informed by Ptera Software's results. Despite these limitations, Ptera Software has proved to be a powerful tool for simulating the dauntingly complex aerodynamics of flapping wing flight. It has the additional benefits of being free to use, open-source, and designed to simulate flapping wings out of the box. For these reasons, Ptera Software is an excellent choice for the research and analysis of volant locomotion and ornithopter development.

References

Binder, S., Wildschek, A., & De Breuker, R. (2017). EXTENSION OF THE CONTINUOUS TIME UNSTEADY VORTEX LATTICE METHOD FOR ARBITRARY MOTION, CONTROL SURFACE DEFLECTION AND INDUCED DRAG CALCULATION. In *International Forum on Aeroelasticity and Structural Dynamics IFASD*.

Fritz, T. E., & Long, L. N. (2004). Object-Oriented Unsteady Vortex Lattice Method for Flapping Flight. *JOURNAL OF AIRCRAFT*, *41*(6), 1275–1290. https://doi.org/10.2514/1.7357

Gardiner, J., Razak, N. A., Dimitriadis, G., Tickle, P., Codd, J. R., & Nudds, R. L. (2013). *Simulation of Bird Wing Flapping Using the Unsteady Vortex Lattice Method*. https://www.researchgate.net/publication/256095328

Guidelines for QFLR5 v0.03 XFLR5 Analysis of foils and wings operating at low Reynolds numbers. (2009).

Katz, J., & Plotkin, A. (2001). Low-Speed Aerodynamics. In *Low-Speed Aerodynamics*. Cambridge University Press. https://doi.org/10.1017/cbo9780511810329

Lambert, T., Abdul Razak, N., & Dimitriadis, G. (2017). Vortex Lattice Simulations of Attached and Separated Flows around Flapping Wings. *Aerospace*, *4*(22). https://doi.org/10.3390/aerospace4020022

Lambert, T., & Dimitriadis, G. (2014). *Modeling of aerodynamic forces in flapping flight with the Unsteady Vortex Lattice Method*.

Nguyen, A. T., Kim, J.-K., Han, J.-S., & Han, J.-H. (2016). Extended Unsteady Vortex-Lattice Method for Insect Flapping Wings. *Journal of Aircraft*, *53*(6), 1709–1718. https://doi.org/10.2514/1.C033456

Roccia, B. A., Preidikman, S., Massa, J. C., & Mook, D. T. (2013). Modified unsteady vortex-lattice method to study flapping wings in hover flight. *AIAA Journal*, *51*(11), 2628–2642. https://doi.org/10.2514/1.J052262

Rohatgi, A. (2020). *Webplotdigitizer: Version 4.4*. https://automeris.io/WebPlotDigitizer

Shyy, W., Lian, Y., Tang, J., Viieru, D., & Liu, H. (2007). Aerodynamics of low reynolds number flyers. In *Aerodynamics Of Low Reynolds Number Flyers* (Vol. 9780521882). Cambridge University Press. https://doi.org/10.1017/CBO9780511551154

Yeo, D., Atkins, E. M., Bernal, L. P., & Shyy, W. (2012). Experimental investigation of the pressure, force, and torque characteristics of a rigid flapping wing. *50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. https://doi.org/10.2514/6.2012-849

Yeo, D., Atkins, E. M., & Shyy, W. (2011). *Experimental and Analytical Pressure Characterization of a Rigid Flapping Wing for Ornithopter Development*. http://paparazzi.enac.fr/wiki/Main_Page

Appendix A: Validation Source Code

```
"""This script runs a validation case of Ptera Software's UVLM.

I first emulate the geometry and kinematics of a flapping robotic test
stand from "Experimental and Analytical Pressure
Characterization of a Rigid Flapping Wing for Ornithopter Development" by
Derrick Yeo, Ella M. Atkins, and Wei Shyy.
Then, I run the UVLM simulation of an experiment from this paper. Finally,
I compare the simulated results to the
published experimental results.

WebPlotDigitizer, by Ankit Rohatgi, was used to extract data from Yeo et
al., 2011.

More information can be found in my accompanying report: "Validating an
Open-Source UVLM
Solver for Analyzing Flapping Wing Flight: An Experimental Approach."
"""

# Import Python's math package.
import math

# Import Numpy and MatPlotLib's PyPlot package.
import matplotlib.pyplot as plt
import numpy as np

# Import Ptera Software
import pterasoftware as ps

# Set the given characteristics of the wing in meters.
half_span = 0.213
chord = 0.072

# Set the given forward flight velocity in meters per second.
validation_velocity = 2.9

# Set the given angle of attack in degrees. Note: If you analyze a
# different operating point where this is not zero,
# you need to modify the code to rotate the experimental lift into the wind
# axes.
validation_alpha = 0

# Set the given flapping frequency in Hertz.
validation_flapping_frequency = 3.3

# This wing planform has a rounded tip so the outermost wing cross section
# needs to be inset some amount. This value is
# in meters.
tip_inset = 0.005

# Import the extracted coordinates from the paper's diagram of the
# planform. The resulting array is of the form
```

```
# [spanwise coordinate, chordwise coordinate], and is ordered from the
# leading edge root, to the tip, to the trailing
# edge root. The origin is the trailing edge root point. The positive
# spanwise axis extends from root to tip and the
# positive chordwise axis from trailing edge to leading edge. The
# coordinates are in millimeters.
planform_coords = np.genfromtxt(
    "Extracted Planform Coordinates.csv", delimiter=","
)

# Convert the coordinates to meters.
planform_coords = planform_coords / 1000

# Set the origin to the leading edge root point.
planform_coords = planform_coords - np.array([0, chord])

# Switch the sign of the chordwise coordinates.
planform_coords = planform_coords * np.array([1, -1])

# Swap the axes to the form [chordwise coordinate, spanwise
# coordinate]. The coordinates are now in the geometry
# frame projected on the XY plane.
planform_coords[:, [0, 1]] = planform_coords[:, [1, 0]]

# Find the index of the point where the planform x-coordinate equals the
# half span.
tip_index = np.where(planform_coords[:, 1] == half_span)[0][0]

# Using the tip index, split the coordinates into two arrays of leading
# and trailing edge coordinates.
leading_coords = planform_coords[:tip_index, :]
trailing_coords = np.flip(planform_coords[tip_index:, :], axis=0)

# Set the number of flap cycles to run the simulation for. The converged
# result is 3 flaps.
num_flaps = 3

# Set the number of chordwise panels. The converged
# result is 5 panels.
num_chordwise_panels = 5

# Set the number of sections to map on each wing half. There will be this
# number +1 wing cross sections per wing half.
# The converged result is 18 spanwise sections.
num_spanwise_sections = 18

# Set the chordwise spacing scheme for the panels. This is set to uniform,
# as is standard for UVLM simulations.
chordwise_spacing = "uniform"

# Calculate the spanwise difference between the wing cross sections.
spanwise_step = (half_span - tip_inset) / num_spanwise_sections
```

```
# Define four arrays to hold the coordinates of the front and back points
# of each section's left and right wing cross
# sections.
front_left_vertices = np.zeros((num_spanwise_sections, 2))
front_right_vertices = np.zeros((num_spanwise_sections, 2))
back_left_vertices = np.zeros((num_spanwise_sections, 2))
back_right_vertices = np.zeros((num_spanwise_sections, 2))

# Iterate through the locations of the future sections to populate the wing
# cross section coordinates.
for spanwise_loc in range(num_spanwise_sections):
    # Find the y coordinates of the vertices.
    front_left_vertices[spanwise_loc, 1] = spanwise_loc * spanwise_step
    back_left_vertices[spanwise_loc, 1] = spanwise_loc * spanwise_step
    front_right_vertices[spanwise_loc, 1] = (
        spanwise_loc + 1
    ) * spanwise_step
    back_right_vertices[spanwise_loc, 1] = (spanwise_loc + 1) * spanwise_step

    # Interpolate between the leading edge coordinates to find the x-
    # coordinate of the front left vertex.
    front_left_vertices[spanwise_loc, 0] = np.interp(
        spanwise_loc * spanwise_step,
        leading_coords[:, 1],
        leading_coords[:, 0],
    )

    # Interpolate between the trailing edge coordinates to find the x-
    # coordinate of the back left vertex.
    back_left_vertices[spanwise_loc, 0] = np.interp(
        spanwise_loc * spanwise_step,
        trailing_coords[:, 1],
        trailing_coords[:, 0],
    )

    # Interpolate between the leading edge coordinates to find the x-
    # coordinate of the front right vertex.
    front_right_vertices[spanwise_loc, 0] = np.interp(
        (spanwise_loc + 1) * spanwise_step,
        leading_coords[:, 1],
        leading_coords[:, 0],
    )

    # Interpolate between the trailing edge coordinates to find the x-
    # coordinate of the back right vertex.
    back_right_vertices[spanwise_loc, 0] = np.interp(
        (spanwise_loc + 1) * spanwise_step,
        trailing_coords[:, 1],
        trailing_coords[:, 0],
    )

# Define an empty list to hold the wing cross sections.
validation_airplane_wing_cross_sections = []
```

```
# Iterate through the wing cross section vertex arrays to create the wing
# cross section objects.
for i in range(num_spanwise_sections):

    # Get the left wing cross section's vertices at this position.
    this_front_left_vertex = front_left_vertices[i, :]
    this_back_left_vertex = back_left_vertices[i, :]

    # Get this wing cross section's leading and trailing edge x coordinates.
    this_x_le = this_front_left_vertex[0]
    this_x_te = this_back_left_vertex[0]

    # Get this wing cross section's leading edge y coordinate.
    this_y_le = this_front_left_vertex[1]

    # Calculate this wing cross section's chord.
    this_chord = this_x_te - this_x_le

    # Define this wing cross section object.
    this_wing_cross_section = ps.geometry.WingCrossSection(
        x_le=this_x_le,
        y_le=this_y_le,
        chord=this_chord,
        airfoil=ps.geometry.Airfoil(name="naca0000",),
        num_spanwise_panels=1,
    )

    # Append this wing cross section to the list of wing cross
    # sections.
    validation_airplane_wing_cross_sections.append(this_wing_cross_section)

    # Check if this the last section.
    if i == num_spanwise_sections - 1:
        # If so, get the right wing cross section vertices at this position.
        this_front_right_vertex = front_right_vertices[i, :]
        this_back_right_vertex = back_right_vertices[i, :]

        # Get this wing cross section's leading and trailing edge x-
        # coordinates.
        this_x_le = this_front_right_vertex[0]
        this_x_te = this_back_right_vertex[0]

        # Get this wing cross section's leading edge y-coordinate.
        this_y_le = this_front_right_vertex[1]

        # Calculate this wing cross section's chord.
        this_chord = this_x_te - this_x_le

        # Define this wing cross section object.
        this_wing_cross_section = ps.geometry.WingCrossSection(
            x_le=this_x_le,
            y_le=this_y_le,
```

```
            chord=this_chord,
            airfoil=ps.geometry.Airfoil(name="naca0000",),
            num_spanwise_panels=1,
        )

        # Append this wing cross section to the list of wing cross
        # sections.
        validation_airplane_wing_cross_sections.append(
            this_wing_cross_section
        )

# Define the validation airplane object.
validation_airplane = ps.geometry.Airplane(
    wings=[
        ps.geometry.Wing(
            symmetric=True,
            wing_cross_sections=validation_airplane_wing_cross_sections,
            chordwise_spacing=chordwise_spacing,
            num_chordwise_panels=num_chordwise_panels,
        ),
    ],
)

# Delete the extraneous pointer.
del validation_airplane_wing_cross_sections

# Initialize an empty list to hold each wing cross section movement object.
validation_wing_cross_section_movements = []

# Define the first wing cross section movement, which is stationary.
first_wing_cross_section_movement = ps.movement.WingCrossSectionMovement(
    base_wing_cross_section=validation_airplane.wings[0].wing_cross_sections[
        0
    ],
)

# Append the first wing cross section movement object to the list.
validation_wing_cross_section_movements.append(
    first_wing_cross_section_movement
)

# Delete the extraneous pointer.
del first_wing_cross_section_movement


def validation_geometry_sweep_function(time):
    """ This function takes in the time during a flap cycle and returns the
    flap angle in degrees. It uses the flapping
    frequency defined in the encompassing script, and is based on a
    fourth-order Fourier series. The coefficients were
    calculated by the authors of Yeo et al., 2011.

    :param time: float or 1D array of floats
```

```
        This is a single time or an array of time values at which to
        calculate the flap angle. The units are seconds.
    :return flap_angle: float or 1D array of floats
        This is a single flap angle or an array of flap angle values at the
        inputted time value or values. The units are
        degrees.
    """

    # Set the Fourier series coefficients and the flapping frequency.
    a_0 = 0.0354
    a_1 = 4.10e-5
    b_1 = 0.3793
    a_2 = -0.0322
    b_2 = -1.95e-6
    a_3 = -8.90e-7
    b_3 = -0.0035
    a_4 = 0.00046
    b_4 = -3.60e-6
    f = validation_flapping_frequency

    # Calculate and return the flap angle(s).
    flap_angle = (
        a_0
        + a_1 * np.cos(1 * f * time)
        + b_1 * np.sin(1 * f * time)
        + a_2 * np.cos(2 * f * time)
        + b_2 * np.sin(2 * f * time)
        + a_3 * np.cos(3 * f * time)
        + b_3 * np.sin(3 * f * time)
        + a_4 * np.cos(4 * f * time)
        + b_4 * np.sin(4 * f * time)
    ) / 0.0174533
    return flap_angle


def normalized_validation_geometry_sweep_function_rad(time):
    """ This function takes in the time during a flap cycle and returns the
    flap angle in radians. It uses a normalized
    flapping frequency of 1 Hertz, and is based on a fourth-order Fourier
    series. The coefficients were calculated by the
    authors of Yeo et al., 2011.

    :param time: float or 1D array of floats
        This is a single time or an array of time values at which to
        calculate the flap angle. The units are seconds.
    :return flap_angle: float or 1D array of floats
        This is a single flap angle or an array of flap angle values at the
        inputted time value or values. The units are
        radians.
    """

    # Set the Fourier series coefficients.
    a_0 = 0.0354
```

```
    a_1 = 4.10e-5
    b_1 = 0.3793
    a_2 = -0.0322
    b_2 = -1.95e-6
    a_3 = -8.90e-7
    b_3 = -0.0035
    a_4 = 0.00046
    b_4 = -3.60e-6

    # Calculate and return the flap angle(s).
    flap_angle = (
        a_0
        + a_1 * np.cos(1 * time)
        + b_1 * np.sin(1 * time)
        + a_2 * np.cos(2 * time)
        + b_2 * np.sin(2 * time)
        + a_3 * np.cos(3 * time)
        + b_3 * np.sin(3 * time)
        + a_4 * np.cos(4 * time)
        + b_4 * np.sin(4 * time)
    )
    return flap_angle


# Iterate through each of the wing cross sections.
for j in range(1, num_spanwise_sections + 1):
    # Define the wing cross section movement for this wing cross section. The
    # amplitude and period are both set to one because
    # the true amplitude and period are already accounted for in the custom
    # sweep function.
    this_wing_cross_section_movement = ps.movement.WingCrossSectionMovement(
        base_wing_cross_section=validation_airplane.wings[
            0
        ].wing_cross_sections[j],
        sweeping_amplitude=1,
        sweeping_period=1,
        sweeping_spacing="custom",
        custom_sweep_function=validation_geometry_sweep_function,
    )

    # Append this wing cross section movement to the list of wing cross
    # section movements.
    validation_wing_cross_section_movements.append(
        this_wing_cross_section_movement
    )

# Define the wing movement object that contains the wing cross section
# movements.
validation_main_wing_movement = ps.movement.WingMovement(
    base_wing=validation_airplane.wings[0],
    wing_cross_sections_movements=validation_wing_cross_section_movements,
)
```

```
# Delete the extraneous pointer.
del validation_wing_cross_section_movements

# Define the airplane movement that contains the wing movement.
validation_airplane_movement = ps.movement.AirplaneMovement(
    base_airplane=validation_airplane,
    wing_movements=[validation_main_wing_movement,],
)

# Delete the extraneous pointers.
del validation_airplane
del validation_main_wing_movement

# Define an operating point corresponding to the conditions of the
# validation study.
validation_operating_point = ps.operating_point.OperatingPoint(
    alpha=validation_alpha, velocity=validation_velocity,
)

# Define an operating point movement that contains the operating point.
validation_operating_point_movement = ps.movement.OperatingPointMovement(
    base_operating_point=validation_operating_point,
)

# Delete the extraneous pointer.
del validation_operating_point

# Calculate the period of this case's flapping motion. The units are in
# seconds.
validation_flapping_period = 1 / validation_flapping_frequency

# Calculate the time step (in seconds) so that the area of the wake ring
# vortices roughly equal the area of the bound
# ring vortices.
validation_delta_time = (
    validation_airplane_movement.base_airplane.c_ref
    / num_chordwise_panels
    / validation_velocity
)

# Calculate the number of steps required for the wing to have flapped the
# prescribed number of times.
validation_num_steps = math.ceil(
    num_flaps / validation_flapping_frequency / validation_delta_time
)

# Define the overall movement.
validation_movement = ps.movement.Movement(
    airplane_movement=validation_airplane_movement,
    operating_point_movement=validation_operating_point_movement,
    num_steps=validation_num_steps,
    delta_time=validation_delta_time,
)
```

```
# Delete the extraneous pointers.
del validation_airplane_movement
del validation_operating_point_movement

# Define the validation problem.
validation_problem = ps.problems.UnsteadyProblem(
    movement=validation_movement,
)

# Delete the extraneous pointer.
del validation_movement

# Define the validation solver.
validation_solver =
ps.unsteady_ring_vortex_lattice_method.UnsteadyRingVortexLatticeMethodSolver(
    unsteady_problem=validation_problem,
)

# Delete the extraneous pointer.
del validation_problem

# Define the position of the coordinates of interest and the area of their
# rectangles. These values were extracted by
# digitizing the figures in Yeo et al., 2011.
blue_trailing_point_coords = [0.060, 0.036]
blue_trailing_area = 0.072 * 0.024
blue_middle_point_coords = [0.036, 0.036]
blue_middle_area = 0.072 * 0.024
blue_leading_point_coords = [0.012, 0.036]
blue_leading_area = 0.072 * 0.024
orange_trailing_point_coords = [0.05532, 0.107]
orange_trailing_area = 0.07 * 0.02112
orange_middle_point_coords = [0.0342, 0.107]
orange_middle_area = 0.07 * 0.02112
orange_leading_point_coords = [0.01308, 0.107]
orange_leading_area = 0.07 * 0.02112
green_trailing_point_coords = [0.04569, 0.162825]
green_trailing_area = 0.04165 * 0.015
green_middle_point_coords = [0.03069, 0.176]
green_middle_area = 0.06565 * 0.015
green_leading_point_coords = [0.01569, 0.1775]
green_leading_area = 0.071 * 0.015

# Run the validation solver. This validation study was run using a
# prescribed wake.
validation_solver.run(prescribed_wake=True)

# Call the software's animate function on the solver. This produces a GIF.
# The GIF is saved in
# the same directory as this script. Press "q," after orienting the view,
# to begin the animation.
ps.output.animate(
```

```
    # Set the unsteady solver to the one we just ran.
    unsteady_solver=validation_solver,
    # Show the pressures in the animation.
    show_delta_pressures=True,
    # Set this value to False to hide the wake vortices in the animation.
    show_wake_vortices=True,
)

# Create a variable to hold the time in seconds at each of the simulation's
# time steps.
times = np.linspace(
    0,
    validation_num_steps * validation_delta_time,
    validation_num_steps,
    endpoint=False,
)

# Discretize the time period of the final flap analyzed into 100 steps.
# Store this to an array.
final_flap_times = np.linspace(
    validation_flapping_period * (num_flaps - 1),
    validation_flapping_period * num_flaps,
    100,
    endpoint=False,
)

# Discretize the normalized flap cycle times into 100 steps. Store this to
# an array.
normalized_times = np.linspace(0, 1, 100, endpoint=False)

# Pull the experimental pressure vs. time histories from the digitized
# data. These data sets are stored in CSV files
# in the same directory as this script. The pressure units used are inAq
# and time units are normalized flap cycle times
# from 0 to 1.
exp_blue_trailing_point_pressures = np.genfromtxt(
    "Blue Trailing Point Experimental Pressures.csv", delimiter=","
)
exp_blue_middle_point_pressures = np.genfromtxt(
    "Blue Middle Point Experimental Pressures.csv", delimiter=","
)
exp_blue_leading_point_pressures = np.genfromtxt(
    "Blue Leading Point Experimental Pressures.csv", delimiter=","
)
exp_orange_trailing_point_pressures = np.genfromtxt(
    "Orange Trailing Point Experimental Pressures.csv", delimiter=","
)
exp_orange_middle_point_pressures = np.genfromtxt(
    "Orange Middle Point Experimental Pressures.csv", delimiter=","
)
exp_orange_leading_point_pressures = np.genfromtxt(
    "Orange Leading Point Experimental Pressures.csv", delimiter=","
)
```

```
exp_green_trailing_point_pressures = np.genfromtxt(
    “Green Trailing Point Experimental Pressures.csv”, delimiter=”,”
)
exp_green_middle_point_pressures = np.genfromtxt(
    "Green Middle Point Experimental Pressures.csv", delimiter=","
)
exp_green_leading_point_pressures = np.genfromtxt(
    “Green Leading Point Experimental Pressures.csv”, delimiter=”,”
)

# Interpolate the experimental pressure data to ensure that they all
# reference the same normalized time scale.
exp_blue_trailing_point_pressures_norm = np.interp(
    normalized_times,
    exp_blue_trailing_point_pressures[:, 0],
    exp_blue_trailing_point_pressures[:, 1],
)
exp_blue_middle_point_pressures_norm = np.interp(
    normalized_times,
    exp_blue_middle_point_pressures[:, 0],
    exp_blue_middle_point_pressures[:, 1],
)
exp_blue_leading_point_pressures_norm = np.interp(
    normalized_times,
    exp_blue_leading_point_pressures[:, 0],
    exp_blue_leading_point_pressures[:, 1],
)
exp_orange_trailing_point_pressures_norm = np.interp(
    normalized_times,
    exp_orange_trailing_point_pressures[:, 0],
    exp_orange_trailing_point_pressures[:, 1],
)
exp_orange_middle_point_pressures_norm = np.interp(
    normalized_times,
    exp_orange_middle_point_pressures[:, 0],
    exp_orange_middle_point_pressures[:, 1],
)
exp_orange_leading_point_pressures_norm = np.interp(
    normalized_times,
    exp_orange_leading_point_pressures[:, 0],
    exp_orange_leading_point_pressures[:, 1],
)
exp_green_trailing_point_pressures_norm = np.interp(
    normalized_times,
    exp_green_trailing_point_pressures[:, 0],
    exp_green_trailing_point_pressures[:, 1],
)
exp_green_middle_point_pressures_norm = np.interp(
    normalized_times,
    exp_green_middle_point_pressures[:, 0],
    exp_green_middle_point_pressures[:, 1],
)
exp_green_leading_point_pressures_norm = np.interp(
```

```
    normalized_times,
    exp_green_leading_point_pressures[:, 0],
    exp_green_leading_point_pressures[:, 1],
)

# Find the normal force time history on each of the experimental panels in
# Newtons.
exp_blue_trailing_normal_forces = (
    248.84 * exp_blue_trailing_point_pressures_norm * blue_trailing_area
)
exp_blue_middle_normal_forces = (
    248.84 * exp_blue_middle_point_pressures_norm * blue_middle_area
)
exp_blue_leading_normal_forces = (
    248.84 * exp_blue_leading_point_pressures_norm * blue_leading_area
)
exp_orange_trailing_normal_forces = (
    248.84 * exp_orange_trailing_point_pressures_norm * orange_trailing_area
)
exp_orange_middle_normal_forces = (
    248.84 * exp_orange_middle_point_pressures_norm * orange_middle_area
)
exp_orange_leading_normal_forces = (
    248.84 * exp_orange_leading_point_pressures_norm * orange_leading_area
)
exp_green_trailing_normal_forces = (
    248.84 * exp_green_trailing_point_pressures_norm * green_trailing_area
)
exp_green_middle_normal_forces = (
    248.84 * exp_green_middle_point_pressures_norm * green_middle_area
)
exp_green_leading_normal_forces = (
    248.84 * exp_green_leading_point_pressures_norm * green_leading_area
)

# Convert each experimental panel's normal force time history to a lift
# time history by finding the vertical component
# given the wing's sweep angle at each time step.
exp_blue_trailing_lift_forces = exp_blue_trailing_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
exp_blue_middle_lift_forces = exp_blue_middle_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
exp_blue_leading_lift_forces = exp_blue_leading_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
exp_orange_trailing_lift_forces = exp_orange_trailing_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
exp_orange_middle_lift_forces = exp_orange_middle_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
```

```
exp_orange_leading_lift_forces = exp_orange_leading_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
exp_green_trailing_lift_forces = exp_green_trailing_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
exp_green_middle_lift_forces = exp_green_middle_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)
exp_green_leading_lift_forces = exp_green_leading_normal_forces * np.cos(
    normalized_validation_geometry_sweep_function_rad(normalized_times)
)

# Calculate the net experimental lift. This is the sum of all the lift on
# each of the experimental panels multiplied by
# two (because the experimental panels only cover one of the symmetric wing
# halves).
# Note: this list of lift forces is with
# respect to the body axes. I will later compare it to the simulated lift
# in wind axes. This does not matter because the
# operating point is at zero angle of attack. If the angle of attack is
# changed, the experimental lift forces must be
# rotated to the wind frame before comparison with the simulated lift
# forces.
exp_net_lift_forces = 2 * (
    exp_blue_trailing_lift_forces
    + exp_blue_middle_lift_forces
    + exp_blue_leading_lift_forces
    + exp_orange_trailing_lift_forces
    + exp_orange_middle_lift_forces
    + exp_orange_leading_lift_forces
    + exp_green_trailing_lift_forces
    + exp_green_middle_lift_forces
    + exp_green_leading_lift_forces
)

# Get this solver's problem's airplanes.
airplanes = []
for steady_problem in validation_solver.steady_problems:
    airplanes.append(steady_problem.airplane)

# Initialize matrices to hold the forces and moments at each time step.
sim_net_forces_wind_axes = np.zeros((3, validation_num_steps))

# Iterate through the time steps and add the results to their respective
# matrices.
for step in range(validation_num_steps):
    # Get the airplane at this time step.
    airplane = airplanes[step]
    # Add the total near field forces on the airplane at this time step to
the
    # list of simulated net forces.
    sim_net_forces_wind_axes[
```

```
        :, step
    ] = airplane.total_near_field_force_wind_axes

# Initialize the figure and axes of the experimental versus simulated lift
# plot.
lift_figure, lift_axes = plt.subplots(figsize=(5, 4))

# Get the simulated net lift forces. They are the third row of the net
# forces array.
sim_net_lift_forces_wind_axes = sim_net_forces_wind_axes[2, :]

# Interpolate the simulated net lift forces to find them with respect to the
# normalized final flap time scale.
final_flap_sim_net_lift_forces_wind_axes = np.interp(
    final_flap_times, times, sim_net_lift_forces_wind_axes[:]
)

# Plot the simulated lift forces. The x-axis is set to the normalized
# times, which may seem odd because we just
# interpolated so as to get them in terms of the normalized final flap
# times. But, they are discretized in exactly the
# same way as the normalized times, just horizontally shifted.
lift_axes.plot(
    normalized_times,
    final_flap_sim_net_lift_forces_wind_axes,
    label="Simulated",
    color="#E62128",
    linestyle="solid",
)

# Plot the experimental lift forces.
lift_axes.plot(
    normalized_times,
    exp_net_lift_forces,
    label="Experimental",
    color="#E62128",
    linestyle="dashed",
)

# Label the axis, add a title, and add a legend.
lift_axes.set_xlabel("Normalized Flap Cycle Time",)
lift_axes.set_ylabel("Lift (N)",)
lift_axes.set_title("Simulated and Experimental Lift Versus Time",)
lift_axes.legend()

# Show the figure.
lift_figure.show()

# Delete the extraneous pointers.
del airplanes
del steady_problem
del sim_net_forces_wind_axes
del step
```

```
# Calculate the lift mean absolute error (MAE). The experimental and
# simulated lift comparison here is valid because,
# due to the interpolation steps, the experimental and simulated lifts
# time histories are discretized so that they
# they are with respect to the same time scale.
lift_absolute_errors = np.abs(
    final_flap_sim_net_lift_forces_wind_axes - exp_net_lift_forces
)
lift_mean_absolute_error = np.mean(lift_absolute_errors)

# Print the MAE.
print(
    "\nMean Absolute Error on Lift: "
    + str(np.round(lift_mean_absolute_error, 4))
    + "N"
)

# Calculate the experimental root mean square (RMS) lift.
exp_rms_lift = np.sqrt(np.mean(np.power(exp_net_lift_forces, 2)))

# Print the experimental RMS lift.
print("Experimental RMS Lift: " + str(np.round(exp_rms_lift, 4)) + " N")
```