Summer 2011

Image based computational fluid dynamics modeling to simulate fluid flow around a moving fish

Justin Wayne Hannon
University of Iowa

IMAGE BASED COMPUTATIONAL FLUID DYNAMICS MODELING TO SIMULATE FLUID FLOW AROUND A MOVING FISH

by

Justin Wayne Hannon

A thesis submitted in partial fulfillment
of the requirements for the Master of
Science degree in Civil and Environmental Engineering
in the Graduate College of
The University of Iowa

July 2011

Thesis Supervisors: Professor Larry J. Weber
Adjunct Assistant Professor Justin Garvin

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

_____

MASTER'S THESIS

_____

This is to certify that the Master's thesis of

Justin Wayne Hannon

has been approved by the Examining Committee
for the thesis requirement for the Master of Science
degree in Civil and Environmental Engineering at the July 2011 graduation.

Thesis Committee: _____
Larry J. Weber, Thesis Supervisor

_____
Justin Garvin, Thesis Supervisor

_____
Jacob Odgaard

To Angela

It seems to me that we all look at Nature too much, and live with her too little.
Oscar Wilde
De Profundis

ACKNOWLEDGMENTS

I would like to thank Justin Garvin for all of the support he has given me over the last few years. He is one of the best mentors I have ever had, and I can't say enough good things about him. I'd like to thank Angela Brown for always being there. We became study partners our first semester of college, and I consider her my best friend.

I'd like to thank Larry Weber, who even though he is a very busy guy, always seems to have time to chat and get to know his students. And finally, I'd like to thank the Hydro Research Foundation for funding this project. This would not have been possible without their support.

ABSTRACT

Understanding why fish move the way in which they do has applications far outside of biology. Biological propulsion has undergone millions of years of refinement, far outpacing the capabilities of anything created by man. Research in the areas of unsteady/biological propulsion has been increasing in the last several decades with advances in technology. Researchers are currently conducting experiments using pitching and heaving airfoils, mechanized fish, and numerical fish. However, the surrogate propulsors that are being used in experiments are driven analytically, whereas in this study, a method has been developed to exactly follow the motion of swimming fish.

The research described in this thesis couples the image analysis of swimming fish with computational fluid dynamics to accurately simulate a virtual fish. Videos of two separate fish swimming modes were analyzed. The two swimming modes are termed 'free-stream swimming' and the 'Kármán gait'. Free-stream swimming is how fish swim in a section of water that is free of disturbances, while Kármán gait swimming is how fish swim in the presence of a vortex street. Each swimming mode was paired with two simulation configurations, one that is free of obstructions, and one that contains a vortex street generating D-section cylinder. Data about the efficiency of swimming, power output, and thrust production were calculated during the simulations.

The results showed that the most efficient mode of swimming was the Kármán gait in the presence of a Kármán vortex street. Evidence corroborating this has been found in the literature. The second most efficient means of swimming was found to be free-stream swimming in the absence of obstructions. The other two configurations, which are not observed in experiments, performed very poorly in regard to swimming efficiency.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

CHAPTER I

BACKGROUND AND MOTIVATION

Background

A great deal of research has gone into studying biological propulsion and behavior in recent years. Research in this area has largely focused on how animals generate unsteady lift and thrust, and why they choose to use the locomotion modes they use. A main in interest in biological propulsion is how oscillating bodies are able to generate thrust at much higher efficiencies than rigid bodies. Numerical studies by Shen *et al*. (2003) and mechanical studies by Barret *et al*. (1999) have shown that the probable mechanisms for the increased efficiency of oscillating bodies over rigid bodies is that non-rigid bodies have reduced separation, and the boundary layer is prevented from transitioning to turbulence. However, there are still remaining questions in the areas of biological propulsion and behavior, in particular, the schooling of fish.

An example of successful research in the area of fish behavioral patterns is the Numerical Fish Surrogate (NFS). The NFS uses an Eulerian-Lagrangian Agent Method to simulate fish behavioral patterns on the aggregate scale, and is used to model fish passage through hydropower dams in the Pacific Northwest. In the NFS, basic behavioral assumptions are made about fish sensory response. The model uses hydraulic strain, velocity, and pressure as the main drivers in fish course correction. This method is backed up by the biological sensory mechanisms that fish use to perceive their environments (Goodwin *et al*., 2006). However, the NFS's accuracy could be improved if more quantitative behavioral data was collected and used in the model.

Research that is aimed at quantifying the behavior of fish motion in different flow fields would certainly be of benefit to improve the NFS model. Currently, the unsteady behavior of aquatic species is a poorly understood area. The review by Liao (2007) explains the currently understood mechanisms in which fish behave in perturbing flows.

Much of the research in Liao's review is experimental, carried out using digital particle image velocimetry (DPIV) and electromyography. DPIV is used to examine experimental flow fields in the vicinity of fish swimming, while electromyography is used to determine muscle activity. One the most interesting modes of locomotion described in the review is called the Kármán gait, initially discovered by Liao *et al*. (2003), and further researched by (Beal *et al*., 2006; Cook and Coughlin, 2010; Liao, 2004; Przybilla *et al*., 2010; Taguchi and Liao, 2011).

The Kármán gait is a form of locomotion that fish (in the experimental studies, a rainbow trout) adopt in the presence of a Kármán vortex street. The Kármán vortex street is a wake pattern which forms behind bluff bodies. It consists of pairs of vortices, of alternating sign, being regularly shed from the body. During the Kármán gait, the fish adopts a swimming mode which features large lateral displacements, and a slaloming motion occurs between the vortices shed from the bluff body. Figure 1 shows the slaloming motion in detail. Figure 2 compares free-stream swimming kinematics (flow in an unobstructed flow field) with the Kármán gait. In these figures, it can be seen that the Kármán gait differs significantly from free-stream swimming. It has been proposed that the Kármán gait allows fish to harness energy from the vortices shed from the bluff body in the flow field, decreasing their energy expenditure, and increasing their swimming efficiency. This proposal has been reinforced using electromyography and respirometry.

In Liao's study using electromyography (2004), he found that when fish were swimming in the Kármán gait, they were activating only their anterior muscles. Further examination of the electromyography results showed that the anterior motion occurring during the Kármán gait was used only for course correction and stability, and that the large lateral displacements were being caused by the Kármán vortex street. A study by Beal *el al*. (2006) has shown that a form of the Kármán gait can be observed in euthanized fish in the presence of a vortex street. In this study, euthanized fish were

attached to a Kármán street generating D-section cylinder, using a tow line. They found that the euthanized fish's body would begin to oscillate, propel itself toward the cylinder, enter the suction zone, and then collide with the cylinder. This reinforces the proposed hypothesis that fish using the Kármán gait were using passive propulsion, and only moving for course correction.

In the respirometry studies (Tagachi and Liao, 2011; Cook and Coughlin, 2010), oxygen uptake rates were observed in fish swimming using free-stream kinematics and the Kármán gait. Both studies found that the rate of oxygen uptake was lower in fish using the Kármán gait, than in fish using free-stream kinematics (free stream in this case meaning no vortex shedding coming from a bluff body upstream). Oxygen uptake is correlated to metabolic output, implying that fish using the Kármán gait were reducing their energy output compared to fish using free-stream kinematics.

An important part of the kinematics of the Kármán gait is its possible applications in the explanation of fish schooling. When fish are swimming, they produce an inverse Kármán street, a thrust wake. An inverse Kármán street is identical to a Kármán street, except that the sign of the vortices is reversed. During schooling, fish are observed as in Figure 3. A fish swimming behind, and between two fish, is exposed to a Kármán street. It experiences one half of the street from each of the fish in front of it. This configuration, and current knowledge of the Kármán gait, can help explain why fish school (efficiency gains). Of course, there are also other benefits to schooling (such as safety in numbers).

At this point, there is a moderate amount of evidence showing that the Kármán gait is an efficient form of locomotion for fish swimming within a Kármán vortex street. However, quantitative data is difficult to acquire in experimental testing. DPIV allows one to examine the flow field around swimming fish, but the resolution is quite low, and the tests can be time consuming and expensive. Electromyography can be used to show muscle activity within the fish, but is also quite time consuming, as the electrodes require

surgical implantation. Respirometry can determine oxygen uptake during swimming, but this is the only variable it can measure. In addition, each of the above methods requires an experimental flume, live fish, and costly equipment.

To overcome these obstacles, researchers often employ the use of computational fluid dynamics (CFD) to model real world phenomena. CFD allows one to model fluid flow and achieve a detailed understanding of what is occurring. Research has be published using CFD simulations to model aquatic locomotion, with good results.

Previous Work

Numerical studies of fish and fishlike swimming are numerous. Research on unsteady airfoils (which can be viewed as not unlike fish) has been conducted by Anderson *et al*., (1997), Koochesfahani (1989), Streitlien and Triantafyllou (1998), Schnipper *et al*., Lai and Platzer (1999), and others. In general, these studies have focused on the performance of a pitching and/or heaving airfoil in thrust production, efficiency, and wake structure. The experiments that were performed were both computational and experimental, and were conducted (typically) over a wide range of parameters. In effect, the studies were done in an attempt to determine over which flow regimes and oscillation parameters thrust and efficiency are maximized.

As stated previously, it has been shown that non-rigid bodies can outperform rigid bodies at thrust production and efficiency. Some numerical studies have been conducted to simulate the motion of fish swimming. Borazjani and Sotiropoulos (2008, 2010) used CFD to model both anguilliform (eel like) and carangiform (mackerel like) swimming. In anguilliform swimming, the entire body of the organism oscillates, whereas in carangiform swimming, only the posterior half of the body oscillates. In the paper published in 2010, they evaluated both body shape and kinematics by mixing and matching body form with kinematics (e.g. carangiform body with anguilliform

kinematics). Their findings showed that wake structure was highly dependent on Strouhal number, but not on body form or kinematics.

All of the numerical studies in the literature have relied on mathematically expressions to generate the motion of oscillatory or moving bodies. This may work well for highly idealized conditions, such as free-stream swimming, but highly complex biological gaits and maneuvers may not be able to be modeled analytically.

To remove the uncertainty that occurs during analytical modeling, a method of image analysis based CFD has been developed. Using this method, videos of fish swimming have been broken down, processed, read into a CFD solver, and used to run CFD simulations of two-dimensional swimming fish. This method of analysis can remove the 'guesswork' from the modeling of biological motion, and is highly extensible to 3D.

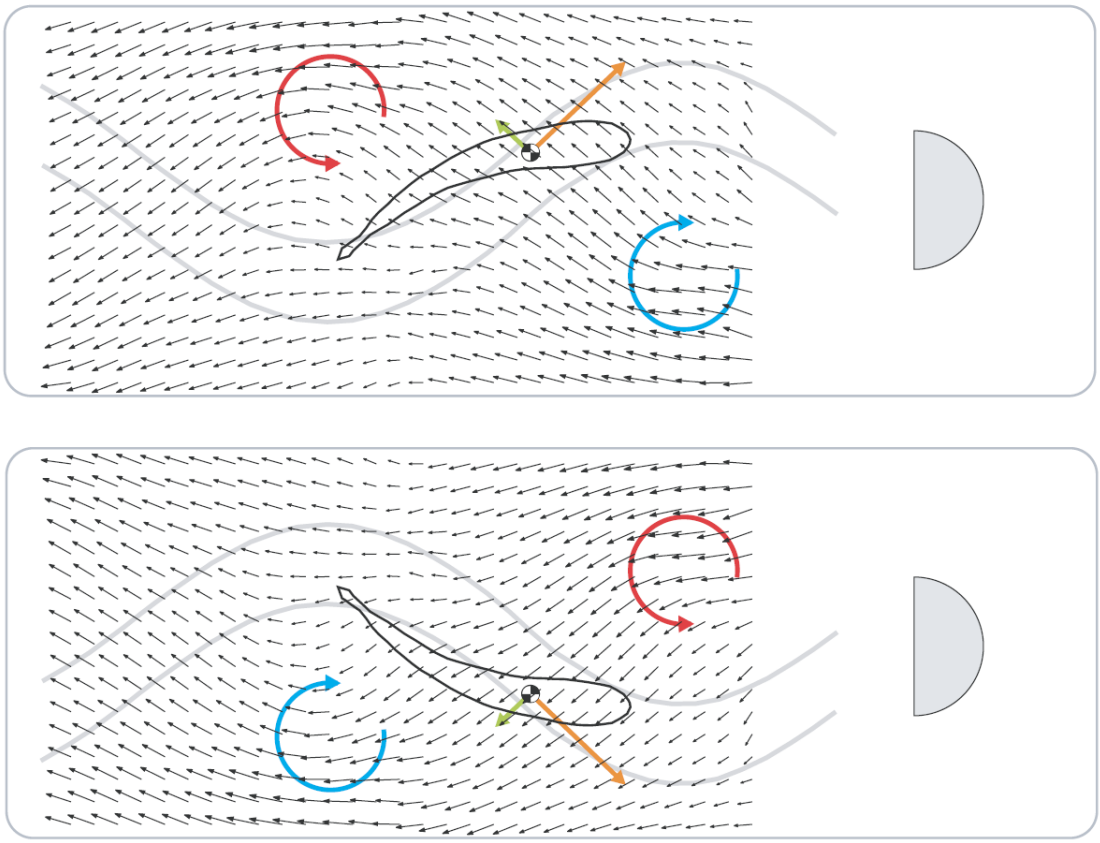


Figure 1: Motion of the Karman gait. The fish slaloms between the vortices shed from the bluff body (in this case a D-section cylinder). It is proposed that the fish does this to harness the energy from the vortices, increasing its swimming efficiency.

Source: Liao, 2007

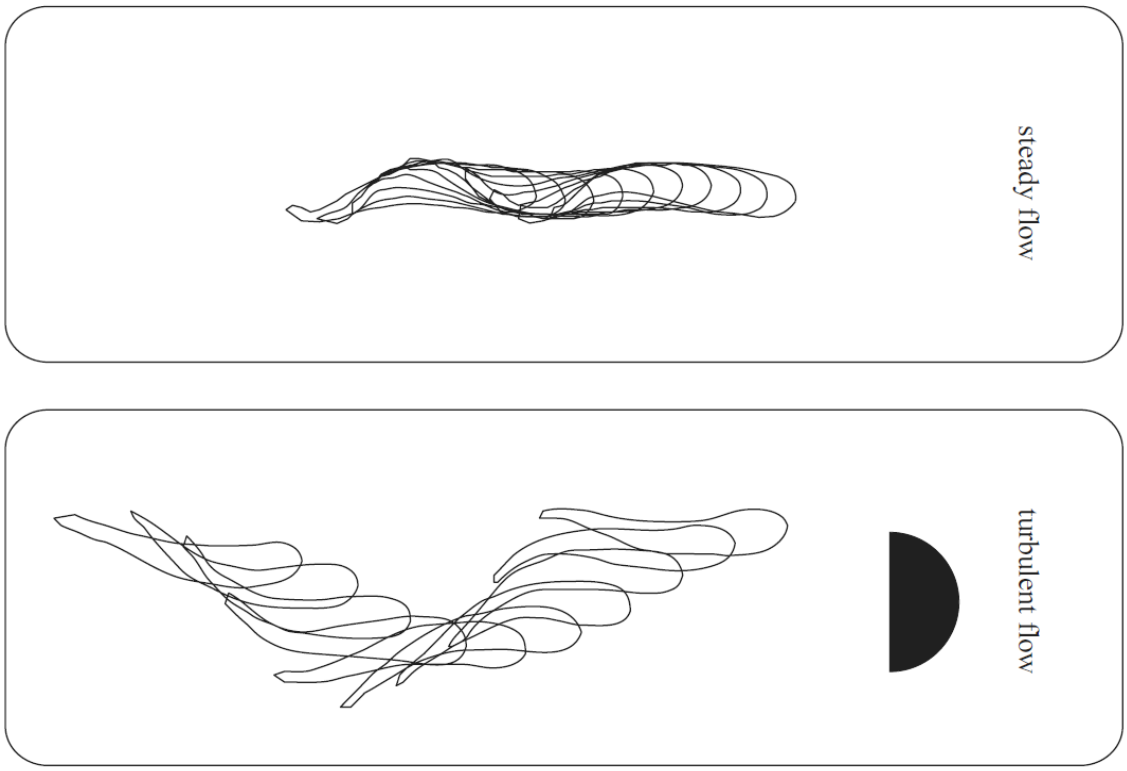


Figure 2: Comparison of free-stream swimming (top) and the Kármán gait (bottom). The large lateral displacement of the Kármán gait is in sharp contrast to the relatively small displacement in free-stream swimming.

Source: Liao, 2007

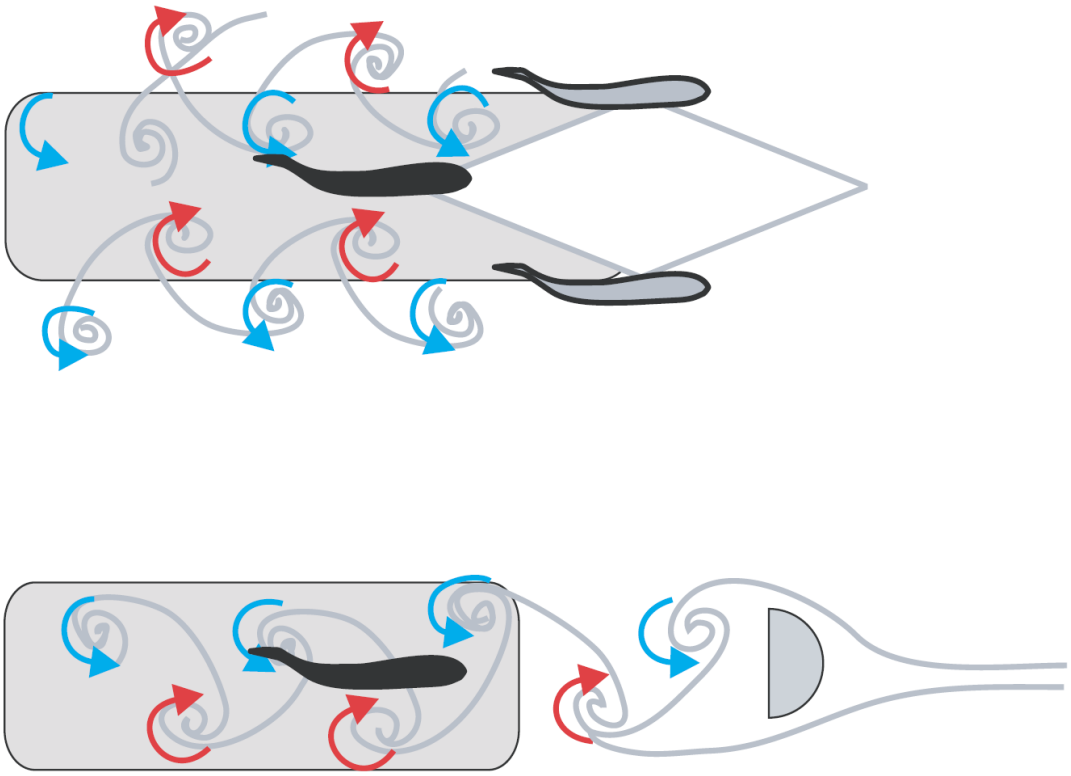


Figure 3: Proposed explanation of schooling using the Kármán gait for increased efficiency. When fish school in this configuration (top), the downstream fish (black) experiences a Kármán vortex street. The pair of fish upstream (grey) produce inverse Kármán streets. However, the fish downstream only experiences one-half of the inverse street from each upstream fish, which is in effect, a Kármán street. This can be approximately by a bluff body (bottom).

Source: Liao, 2007

CHAPTER II

OVERVIEW OF RESEARCH

Introduction

Computational fluid dynamics (CFD) is an important research tool that allowsinvestigators to perform experiments and collect data that may not be possible or practical to collect using traditional physical experiments. The present study aims to simulate multiple configurations of two-dimensional fish swimming in virtual flumes. This work was based off research conducted by Liao *et al*. (2003), who has conducted many experiments in the area of fish locomotion. Liao *et al*.'s findings that the Kármán gait is an efficient means of propulsion under certain conditions were tested using CFD. Four configurations of simulations were carried out:

1. Free-stream kinematics in an empty flume
2. Free-stream kinematics behind a D-section cylinder
3. Kármán gait kinematics in an empty flume
4. Kármán gaitkinematics behind a D-section cylinder

Configurations 1 and 4 above represent a matching of kinematics and flow field that are observed experimentally, while configurations 2 and 3 are not.

It has been proposed that configurations 1 and 4 are observed experimentally because these configurations are efficient means of swimming in their respective flow fields. Liao (2004), Taguchi and Liao (2011), and Cook and Coughlin (2010) have carried out experiments that show configuration 4 provides the fish with greater propulsive efficiency than configuration 1.

The goal of this study was to develop a method to accurately simulate fish motion, and quantify the efficiency at which they swim in various flow conditions. Based on experimental work by Liao (2004, 2011) and Cook and Coughlin (2011), it was believed that Kármán gait swimming paired with a D-section cylinder (configuration 4)

would be the most efficient means of propulsion. The second highest efficiency was expected to be free-stream swimming in an empty flume (configuration 1), as it is an observed swimming gait found in nature. Finally, configurations 2 and 3 were expected to have poor swimming efficiencies, because these are configurations which have not been observed experimentally.

Importance of Work

The simulations that were performed in this study used a method of image analysis to capture the motion of fish swimming in an experimental flume. Previous computational studies (Adkins & Yan, 2006; Borazjani & Sotiropoulos, 2008, 2009) have assigned analytical expressions to simulate fish motion. While some forms of fish locomotion lend themselves well to mathematically prescribed motion (such as free-stream locomotion), others do not. The Kármán gait, which features large amplitude tail-beats and large lateral displacements, would be difficult to describe mathematically. By capturing the exact motions of a fish swimming in an experimental test flume, it is possible to capture all of the details of fish locomotion, eliminating error that may be introduced by a mathematical model. In the future, this method of using image analysis based CFD could be extended to analyze the details of complicated locomotion, and could possibly be applied to non-aquatic species.

Overall Research Strategy

The current study was fairly complex and required the use of many pieces of software, both custom and commercial. A brief overview of the steps required to complete the simulations is as follows:

1. Divide a video of a fish swimming in an experimental test flume into individual frames, each separated by 1/250-s.

2. Run image analysis software on each video frame to generate *x* and *y* coordinates that define the body of the fish. These x and y coordinates are stored in files referred to as "fish data files" (FDFs)
3. Create a numerical grid containing a virtual flume and fish. The initial shape and position of the fish is taken from the data extracted from the first video frame.
4. Run a CFD analysis on the virtual flume and fish. Throughout the simulation, the position and shape of the virtual fish is updated using the coordinates stored in the FDFs.
5. During the simulation, collect data on the force, drag, thrust, and power generated by the swimming fish.

The above steps are fairly general, but give a good idea of the work required to complete the simulations. The remainder of this chapter will focus on steps 1 and 2, with a brief introduction to steps 3 and 4. The details of steps 3 and 4 can be found in Chapter III, and information on step 5 can be found in Chapter IV.

Image Analysis

The videos used in this research were provided by George Lauder at Harvard University (http://www.people.fas.harvard.edu/~glauder/). These videos were originally used for experimental research on fish locomotion using digital particle image velocimetry (DPIV). An example frame from one of the videos can be seen in Figure 4. The videos were collected using a high-speed digital camera that captured data at 250 frames per second. Two videos were used in this study, each showing a rainbow trout, holding station, using a different form of locomotion. One video showed free-stream swimming, while the other showed a fish swimming in the Kármán gait behind a D-section cylinder. In both cases, the free-stream velocity of the water was 0.45-m/s. The experimental flume measured 80-cm long x 28-cm wide x 28-cm deep.

The first step in analyzing the videos consisted of dividing them into individual frames. Each frame had a corresponding time-stamp indicating when that particular frame was displayed. The individual video frames were then used to construct data files containing coordinates of the fish body. The first step in the image analysis algorithm involved locating the head of the fish. This was done using template matching (Garvin *et al*., 2011). Template matching locates the fish's head by using a known image of the fish's head. Each video pixel is statistically compared with the known image (at many orientations and scales) until a likely match is found.

Once the head of the fish was located, thresholding was used to remove artifacts of the flume from the video frame, leaving only the silhouette of the fish body. This was possible because of the high contrast lighting used when the video was captured. The video frame was then converted to a binary image, consisting of pixels that were either black (the fish) or white (the background).

The spine of the fish was identified by using skeletonization. Skeletonization creates a line, a single pixel wide, equidistant from the boundaries of the fish body. During skeletonization, small branches (lines extending from the spine) often form, and were removed using dilation and thinning (Garvin *et al*., 2011). Once the location of the spine and head were determined, a smoothing spline was fit to them. The splines were constrained to a constant length across all of the video frames, to prevent unintended longitudinal deformation.

The points defining the fish body were located by extending evenly spaced lines perpendicular to the spline of the fish spine (Figure 5). The intersection of the perpendicular lines with the exterior of the thresholded fish body is where the points were recorded. The image analysis software wrote out a total of 197 and 196 points (for both images?) along the fish body, for the free-stream video and Kármán gait video, respectively. A step-by-step illustration of the image analysis process can be seen in Figure 6.

If more points on the fish body were needed (e.g., the numerical grid needed refining near the fish body), multiple passes of linear interpolation were performed on the boundary points until the desired level of refinement had been reached.Once the image analysis data is collected, it is used in the CFD analysis to drive the motion of the simulated fish.

Overview of Computational Work

Acquiring data files that describe the motion of the fish was the first step in this study. The second step was to generate a numerical grid for CFD analysis. This was performed using the commercial software Pointwise. This program offers a large amount of flexibility in grid generation, and it was fairly simple to generate the numerical grid.

The next step of the simulation was to conduct the CFD analysis itself. This was done using the commercial CFD solver FLUENT. FLUENT is a feature rich program that offers the user many turbulence models, discretization schemes, physics models, etc. FLUENT's dynamic meshing capabilities were coupled with a user-defined function to generate the motion of the fish swimming. Before fish locomotion was initiated, the simulations were run to steady-state. This ensured that initial transient effects would not have an impact on the results. A total of ten fish simulations were completed, consisting of the four configurations listed in the introduction. Four simulations were completed for each of the cases in which a D-section cylinder was present. One simulation was completed for each of the cases in which the flume only contained a fish. The reasoning behind this will be described in Chapter IV.

Data Acquisition and Post-processing

The goal of this research was to quantify the efficiency of swimming fish, for two distinct methods of locomotion. The calculation of swimming efficiency required data that is not directly accessible from the FLUENT user interface. During the simulations, the relevant data was collected using a user-defined function. The user-defined function

then made the necessary calculations, and wrote the data to a file. This occurred once every time-step, after the fish had been put in to motion. The details of this user-defined function can be found in Chapter IV.

During the simulations, FLUENT case and data files were exported every 0.01-s. These files were used to generate contour plots and videos of vorticity, which can be seen in Chapter IV, and with the supplemental material, respectively. Examining the vorticity in the simulations provided a way to examine why certain configurations provided efficiency gains.

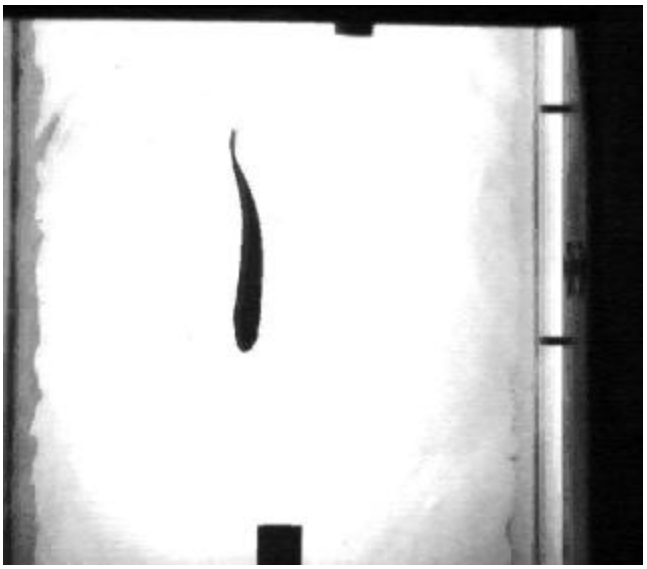


Figure 4: Example of an extracted image from a video showing free-stream swimming.

Source: George Lauder, http://www.people.fas.harvard.edu/~glauder/

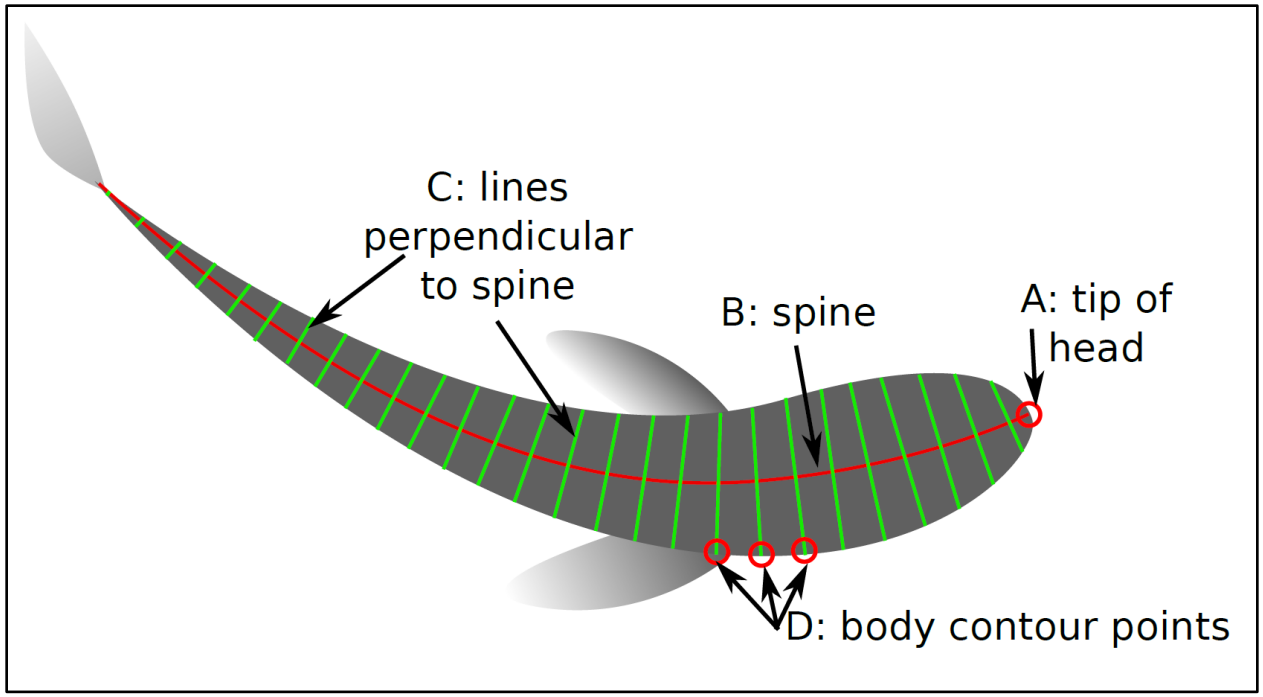


Figure 5: Image showing the features that are identified during the image analysis process. (A) The tip of the head is located using a template matching method. (B) The spine is found through skeletonizing the thresholded image of the fish silhouette. The skeletonized spine is then fit to a spline. (C) Equally spaced lines are connected perpendicular to the splined spine. (D) The intersection of the thresholded fish body and the lines perpendicular to spine produces the body contour points, they points which govern the motion of the fish during CFD analysis.

Source: Garvin *et al*., 2011

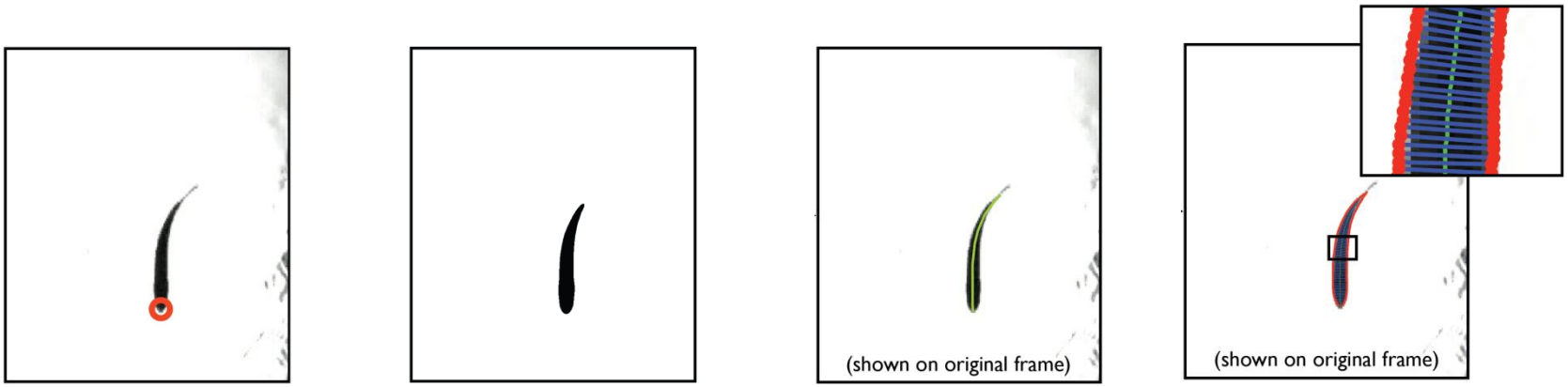


Figure 6: Steps of image analysis. (A) The tip of the fish head is located. (B) The image is thresholded, leaving only the fish silhouette. (C) The spline of the spine is created. (D) Lines perpendicular to the spine spline are drawn, and fish boundary points are located.

Source: Garvin *et al*., 2011

CHAPTER III

COMPUTATIONAL MODEL, NUMERICAL GRID, AND SOLUTION STRATEGY

Introduction

When carrying out simulations of Karman gait and free-stream locomotion, the goal was to mimic the experiments performed byLiao*et al*. (2003) as closely as possible. Liao*et al*.'s experimental set-up consisted of a 28-cm x 28-cm x 80-cm flow tank, in which a 5-cm D-section cylinder was placed (Figure 7). A 10-cm rainbow trout was placed inside the experimental flume, and its locomotion was observed using high-speed digital particle image velocimetry (DPIV).

To create an accurate simulation of Liao*et al*.'s experimental work, it was necessary to use the correct flow solver physics. The flow solver that was used was the commercial CFD program, FLUENT v12. FLUENT is highly extensible through the use of user-defined functions (UDFs). UDFs are small programs written in C that use FLUENTs powerful data-access libraries and allow the user to extend FLUENTs capabilities. FLUENT UDFs can be used to generate custom turbulence models, custom data reporting, custom deforming grids, and much more. FLUENT UDFs were used in this study to define the movement of the fish and generate custom data reports.

Domain Geometry

The fluid domain was modeled based on the experiments by Liao*et al*. (2003). It consisted of a two dimensional flume, measuring 80-cm long and 28-cm wide. For some of the simulations, a 5-cm diameter D-section cylinder was placed approximately 15-cm from the upstream inlet of the flume. In all simulations except for validation, a 10-cm fish boundary was placed in the flume 35-cm from the upstream inlet of the flume.

The fluid domain was re-created as closely as possible according to Liao *et al*. (2003), with the only major difference being that the current study was undertaken in two dimensions.

Simulation Physics

Material Properties

Liao *et al*. (2003) used fish swimming in a highly controlled flume. Properties of water are readily available and the default FLUENT material properties for water were used. The relevant properties were:

Density: 998.2 kg/m^3

Dynamic viscosity: 0.001003kg/m·s

These properties are fairly standard, and are similar to what a fish swimming in a natural environment may encounter.

Comparison of Turbulence Models

The selection of turbulence model was fairly complicated. There are many available turbulence models available in FLUENT, and each model has different intended uses. The models range from the most simple, no turbulence model (laminar flow), to the most complex, large eddy simulation (LES) (ANSYS Inc., 2009). When selecting a turbulence model, one has to weigh simulation accuracy with the computational resources that will be required. Three turbulence models were tested for their simulation accuracy and computational intensity. These tested models were the unsteady Reynolds-averaged Navier-Stokes model (URANS), detached-eddy simulation (DES), and large-eddy simulation.

Unsteady Reynolds-averaged Navier-Stokes

The URANS model's greatest advantage is that it is one of the least computationally expensive turbulence models. However, URANS's computational

cheapness comes at a price. URANS is unable to resolve many spatial and temporal turbulence scales, and thus, they must be modeled. Results from URANS simulation only resolve flow data from the largest turbulence scales, which can be problematic for some types of simulations (Mittal, 2004).

URANS has been shown to have poor performance in regard to flow separation (Mittal, 2004;ANSYS Inc., 2009). This is due to the fact that the URANS model makes the assumption near-wall flows will have an attached boundary layer. Massive flow separation can still be resolved by URANS, but whether the separation will be accurate cannot be determined *a priori*.

Large-eddy Simulation

Large-eddy simulation (LES) resolves turbulence down to the scale of the mesh. All scales smaller than individual cells (the subgrid-scales) are modeled. This is a large improvement over the URANS model, which modeled all but the largest of scales. LES has been shown to give highly accurate results when compared with experimental data. This accuracy comes at a cost, however. LES is very computationally expensive, with estimates stating that the number of grid points must scale as Re^2(Mittal, 2004). This computational expense currently places LES within the realm of simulations at low Reynolds number, and high-performance computing.

Detached-eddy Simulation

Detached-eddy simulation (DES) was created to reduce the computational expense of LES, while increasing accuracy over URANS. DES can be viewed as a hybrid model, in which LES is used where the grid scale is fine enough (typically in areas of separation), and URANS is used in the boundary layer. The DES approach has worked well (Spalart, 2009), and has been shown to be more accurate than URANS in many simulations.

Selection of Turbulence Model

The turbulence model that was selected was based on knowledge of the models, and how they performed in several preliminary simulations. These preliminary simulations consisted of the stated domain geometry (a 28-cm x 80-cm flume) with only the D-section cylinder present in the fluid. No simulated fish were included in the validation/model-selection simulations.

The URANS simulation used the realizable k-ε turbulence model with enhanced wall treatment. The k-ε turbulence model is a two equation turbulence model in which transport equations for turbulent kinetic energy and dissipation rate are used for closure. The realizable k-ε turbulence model is modified from the standard k-ε model in that it uses a new turbulent viscosity equation, and new transport equation for dissipation rate (Shih *et al*., 1995). These modifications were made to make the standard k-ε model in-line with the physics of turbulent fluids. The realizable k-ε model is better performing in flows that involve adverse pressure gradients and separation. This is why the realizable k-ε model was chosen as the candidate URANS model.

The other models tested were the default DES and LES models available in FLUENT. These models offer very little in the way of customization compared to the URANS models. The DES and LES models are not available by default in 2D FLUENT, and must be enabled with the following command: (rpsetvar 'les-2d? #t)

The performance of the three models was varied. The realizable k-ε model was by far the least computationally expensive model. In a test case that simulated 10-s of flow time, URANS, LES, and DES took 11-hrs, 44.5-hrs, and 25-hrs to finish, respectively. Flow separations that lead to a Kármán street were observed in all of the turbulence models. The agreement with experimental data was significantly better in the k-ε model compared to the DES and LES models (see Figures 8 – 10). The lack of accuracy in the LES and DES results may stem from the test cases only being two dimensional. LES and DES are rarely used in 2D, as turbulence is mathematically a 3D

phenomenon (Tsinober, 2001). A large number of simulations needed to be ran for the study, and computational expense was of great concern. Because of its accuracy and speed, the realizable k-ε model was selected as the turbulence model for future simulations.

Boundary Conditions and Initial Conditions

The boundary conditions in the simulation domain were dictated by the experimental setup in Liao *et al*(2003). The boundary conditions were quite simple, as the domain geometry was quite simple. The domain is characterized by its boundaries (an inlet, outlet, and two walls), along with a fish and D-section cylinder within the fluid. The inlet was specified as a 'velocity inlet', with a fluid velocity of 0.45 m/s, normal to the inlet. The outlet was specified a 'pressure outlet' with zero gage pressure at the outlet. The flume walls, the fish body, and the D-section cylinder were specified as no-slip walls. The fish body was set in to motion using FLUENT UDFs for dynamic motion. This will be discussed in a later section.

The initial conditions in the fluid domain were specified according to the velocity inlet properties. Using the velocity inlet's properties as the initial condition for the fluid decreased the time required for the simulations to reach steady-state.

Numerical Grid

Grid Type Considerations

The computational fluid dynamics grid (or mesh) covers the domain over which the flow solver will find solutions to the governing partial differential equations (PDEs) of the problem. The computational mesh discretizes the solution domain into a large number of individual cells, which may vary in size and shape. This process of discretization is what allows the flow solver to compute the flow field. There are several important considerations to be made when generating a mesh, the most important of

which are the mesh type (structured or unstructured) and the relative size of each computational cell.

Structured grids are discretized using cells which are quadrilaterals (2D) or hexahedra (3D). Because structured grids use only quadrilaterals or hexahedra, neighboring cells can be represented in the computer memory efficiently, leading to a faster flow solution. However, structured grids cannot handle complicated geometries well (especially deforming geometries) without using advanced solution methods (which were not available FLUENT). Structured grids also have the advantage of being rectilinear, which provides some benefit in simulation accuracy (ANSYS Inc., 2009).

Unstructured grids can be composed of any polygon (2D) or polyhedron (3D). The most common cell shape for two-dimensional unstructured grids is a triangle. Unstructured grids do not have the memory advantages that structured grids do, due to their inherent irregularity. The location of every cell must be stored in memory by the flow solver. The advantage of unstructured grids is their arbitrary layout of cells, which allows the grid to follow complex geometries with a high degree of accuracy. In addition, unstructured grids can be dynamically remeshed if the domain boundaries are undergoing large deformations. Remeshing ensures that individual cells do not become overly skewed during deformation of the grid.

Generation of the Numerical Grid

The numerical grid was generated using a Pointwise. Pointwise is a commercial grid generation program that can generate both structured and unstructured grids. The solution domain that needed to be meshed consisted of four walls, a fish body, and (in some cases) a D-section cylinder (see Figure 7). The fish body was a fairly complex shape that would need to undergo large deformations during the simulations. For this reason, it was decided to discretize the flow field using an unstructured, triangular mesh.

The mesh of the flume was built around the coordinate system that was produced by the output of the image analysis software. The image analysis software produced data that featured a random origin and scale (e.g., the center of gravity of the fish would be located at x = 560.3, y = 306.2, and the fish would be 407.3 units long). After reading in the data file describing the fish's initial position into Pointwise, the proper scaling factor was determined, from the knowledge that the fish should be 10-cm long. Knowing the 'true' position of the fish inside flume allowed the construction of the walls, inlet, and outlet. After the geometry was completed, it was scaled in size to the proper length scale, before being read into the flow solver.

The refinement level of the mesh was dictated by the turbulence wall functions that were to be used by the flow solver. In a CFD simulation, the number of cells required to discretize the flow field greatly increases in regions of high gradients. To get around this requirement, turbulence models employ wall functions that do not fully resolve the turbulent flow in the region of the wall. Wall functions reduce the accuracy of the solution, but also reduce the computational complexity of the simulation substantially.

The wall function model used in the present study is termed 'enhanced wall treatment'. The recommended mesh resolution for the use of the enhanced wall treatment model is that the wall y^+ should be on the order of one (ANSYS, Inc., 2009). The calculation of wall y^+ is done using the following equations:

$$y^+ = \frac{u_* y}{\upsilon} \qquad \text{Eq. 1}$$

where

$$u_* = \sqrt{\frac{\tau_w}{\rho}} \qquad \text{Eq. 2}$$

and

$$\tau_w = \mu \frac{\partial u}{\partial y} \qquad \text{Eq. 3}$$

where, μ is dynamic viscosity, u is velocity tangential to the wall, y is the wall normal direction, τ_w is the wall shear stress, ρ is the density of the fluid, u_* is the wall friction velocity, and v is the kinematic viscosity.

This value of wall y^+ can only be roughly estimated *a priori*. Therefore, the numerical grid was iteratively refined until the wall y^+ values at the boundaries, calculated by the flow solver, were close to unity. The number of nodes and cells for each simulation case can be found in Table 1. Close-up images of the free-stream and Kármán gait grids can be seen in Figures 11 and 12, respectively.

Implementation of Grid Motion

The output from the image analysis procedure (see Chapter II) was a series of fish data files (FDFs) that corresponded to Cartesian points along the fish's body. Each data file corresponded to a specific point in simulation time. For example, FDF '0' was the position of the fish at *time = 0-s* and FDF '1' was the position of the fish at *time = 0.004-s*, and so on. The motion of the fish was quite complicated, and would need to be performed using one of FLUENT's UDFs.

The macro (the term FLUENT uses to describe built in libraries) that was used to implement the deforming mesh was the DEFINE_GRID_MOTION dynamic mesh macro. The DEFINE_GRID_MOTION macro allows the user to specify the position of nodes on a boundary through time. The user is not limited in which way the position is specified (ANSYS, Inc., 2009).

The FDFs time-step did not correspond to the fluid flow time-step used by the solver. This required interpolation between FDFs during the simulation.

Interpolation Methods

Several interpolation methods were used in an attempt to smooth out rapid changes in the velocity of the fish boundary nodes. These rapid changes in velocity were most problematic in linear interpolation, reduced in second-order interpolation, and were least noticeable using a cubic Catmull-Rom spline.

Linear Interpolation

Linear interpolation was the first method employed to determine the location of the fish boundary nodes between FDFs. It was the simplest method, and an obvious first choice to use.

$$x^{n}_{interp} = x^{i}_{frame} + \left[t^{n}_{CFD} - t^{i}_{frame}\right]\frac{x^{i+1}_{frame} - x^{i}_{frame}}{t^{i+1}_{frame} - t^{i}_{frame}} \quad \text{Eq. 4}$$

$$y^{n}_{interp} = y^{i}_{frame} + \left[t^{n}_{CFD} - t^{i}_{frame}\right]\frac{y^{i+1}_{frame} - y^{i}_{frame}}{t^{i+1}_{frame} - t^{i}_{frame}} \quad \text{Eq. 5}$$

$$x^{n+1}_{interp} = x^{i}_{frame} + \left[t^{n+1}_{CFD} - t^{i}_{frame}\right]\frac{x^{i+1}_{frame} - x^{i}_{frame}}{t^{i+1}_{frame} - t^{i}_{frame}} \quad \text{for} \quad t^{n+1}_{CFD} \le t^{i+1}_{frame} \quad \text{Eq. 6}$$

$$y^{n+1}_{interp} = y^{i}_{frame} + \left[t^{n+1}_{CFD} - t^{i}_{frame}\right]\frac{y^{i+1}_{frame} - y^{i}_{frame}}{t^{i+1}_{frame} - t^{i}_{frame}} \quad \text{for} \quad t^{n+1}_{CFD} \le t^{i+1}_{frame} \quad \text{Eq. 7}$$

$$x^{n+1}_{interp} = x^{i+1}_{frame} + \left[t^{n+1}_{CFD} - t^{i+1}_{frame}\right]\frac{x^{i+2}_{frame} - x^{i+1}_{frame}}{t^{i+1}_{frame} - t^{i+1}_{frame}} \quad \text{for} \quad t^{n+1}_{CFD} > t^{i+1}_{frame} \quad \text{Eq. 8}$$

$$y^{n+1}_{interp} = y^{i+1}_{frame} + \left[t^{n+1}_{CFD} - t^{i+1}_{frame}\right]\frac{y^{i+2}_{frame} - y^{i+1}_{frame}}{t^{i+2}_{frame} - t^{i+1}_{frame}} \quad \text{for} \quad t^{n+1}_{CFD} > t^{i+1}_{frame} \quad \text{Eq. 9}$$

In the above equations, the *n* and *n+1* indices correspond to the current and following CFD time-step, respectively. In Equations 4, 5, 6 and 7, the *i* and *i+1*indices correspond to the FDFs that precede and follow the interpolated data, respectively. In Equations 8 and 9, the *i+1* and *i+2* indices correspond to the FDFs that precede and follow the interpolated data, respectively. Equations 8 and 9 were used as a special case when the next time step and current time-step used different sets of FDFs (i.e., the current time-step lands exactly at a time corresponding to a FDF).

After the interpolated values had been calculated, the velocities of the fish boundary nodes were calculated using the following equations:

$$u_{interp}^{n} = \frac{x_{interp}^{n+1} - x_{interp}^{n}}{\Delta t_{CFD}} \quad \text{Eq. 10}$$

$$v_{interp}^{n} = \frac{y_{interp}^{n+1} - y_{interp}^{n}}{\Delta t_{CFD}} \quad \text{Eq. 11}$$

where Δt_{CFD} is the current solver time-step size, and *u* and *v* are the *x* and *y* velocities of the boundary nodes, respectively. Once the velocities were calculated, the positions of the nodes were updated as follows:

$$x_{fish,CFD}^{n+1} = x_{fish,CFD}^{n} + u_{interp}^{n}\,\Delta t_{CFD} \quad \text{Eq. 12}$$

$$y_{fish,CFD}^{n+1} = y_{fish,CFD}^{n} + v_{interp}^{n}\,\Delta t_{CFD} \quad \text{Eq. 13}$$

This method did not perform as well as had been hoped, especially at small solver time-steps. When linear interpolation was used, the velocity of each node was constant between each set of FDFs (e.g., node 37 has a velocity of (0.04,-0.2) from time-step 0.012-s to time-step 0.016-s). When the next FDF was used to calculate a new node velocity,a massive acceleration was caused at small time-steps. This caused large fluctuations in the flow field, producing poor results.

Second Order Interpolation

The next type of interpolation that was used was a second-order method. In this method, the velocities of the boundary nodes were interpolated instead of the positions. It was thought that interpolating velocity instead of position would help smooth out the large accelerations that occurred between FDFs. The second-order polynomial coefficients were calculated with the following assumptions (the 'frame' subscript has been dropped for clarity):

$$\frac{x^{i+1} - x^{i}}{t^{i+1} - t^{i}} = a_x\left[t^{i}\right]^2 + b_x\left[t^{i}\right] + c_x \quad \text{Eq. 14}$$

$$\frac{y^{i+1}-y^{i}}{t^{i+1}-t^{i}} = a_y\left[t^{i}\right]^2 + b_y\left[t^{i}\right] + c_y \quad \text{Eq. 15}$$

$$\frac{x^{i+2}-x^{i+1}}{t^{i+2}-t^{i+1}} = a_x\left[t^{i+1}\right]^2 + b_x\left[t^{i+1}\right] + c_x \quad \text{Eq. 16}$$

$$\frac{y^{i+2}-y^{i+1}}{t^{i+2}-t^{i+1}} = a_y\left[t^{i+1}\right]^2 + b_y\left[t^{i+1}\right] + c_y \quad \text{Eq. 17}$$

$$\int_{t^i}^{t^{i+1}} [a_x {t_{CFD}}^2 + b_x t_{CFD} + c_x] = x^{i+1} - x^{i} \quad \text{Eq. 18}$$

$$\int_{t^i}^{t^{i+1}} \left[a_y {t_{CFD}}^2 + b_y t_{CFD} + c_y\right] = y^{i+1} - y^{i} \quad \text{Eq. 19}$$

The first four assumptions state that the velocity at the beginning and end of a set of FDFs should equal the linearly interpolated velocities from above. The last two equations state that the total distance traveled by a node during linear interpolation should be conserved. The coefficients *a*, *b*, and *c*were solved for using a computer algebra system and were found to be:

$$a_x = \frac{3\left[x^{i+2}-2x^{i+1}+x^{i}\right]}{\left[t^{i}-t^{i+1}\right]^3} \quad \text{Eq. 20}$$

$$a_y = \frac{3\left[y^{i+2}-2y^{i+1}+y^{i}\right]}{\left[t^{i}-t^{i+1}\right]^3} \quad \text{Eq. 21}$$

$$b_x = 2\frac{\left(t^{i}+t^{i+1}\right)\left(x^{i+2}-2x^{i+1}+x^{i}\right)}{\left[t^{i}-t^{i+1}\right]^3} \quad \text{Eq. 22}$$

$$b_x = 2\frac{\left(t^{i}+t^{i+1}\right)\left(y^{i+2}-2y^{i+1}+y^{i}\right)}{\left[t^{i}-t^{i+1}\right]^3} \quad \text{Eq. 23}$$

$$c_x = \frac{-2\left(t^{i}t^{i+1}\right)\left(x^{i+2}-3x^{i+1}+2x^{i}\right)+\left(t^{i+1}\right)^2\left(x^{i}-x^{i+1}\right)+\left(t^{i}\right)^2\left(x^{i+1}-x^{i+2}\right)}{\left[t^{i}-t^{i+1}\right]^3} \quad \text{Eq. 24}$$

$$c_y = \frac{-2\left(t^{i}t^{i+1}\right)\left(y^{i+2}-3y^{i+1}+2y^{i}\right)+\left(t^{i+1}\right)^2\left(y^{i}-y^{i+1}\right)+\left(t^{i}\right)^2\left(y^{i+1}-y^{i+2}\right)}{\left[t^{i}-t^{i+1}\right]^3} \quad \text{Eq. 25}$$

These coefficients produced an equation for velocity between FDFs. This velocity equation was used to move the fish boundary in a way very similar to Equations 10 and 11.

The second-order velocity equation was smooth between FDFs, but still not smooth at the time-step between them. Again, large fluctuations in the flow field were observed.

Catmull-Rom Spline

The final interpolation method that was tested was a cubic Catmull-Rom spline. This technique provides smoothness through the transitions between data points (Burger and Burge, 2008), eliminating the large spikes in acceleration and fluctuations in the flow field. In this method, the relevant FDF values are assigned weights based on the current solver time-step. The weights were calculated using the following expression:

$$t_j^n = i + \frac{t_{CFD}^n - t_{frame}^i}{t_{frame}^{i+1} - t_{frame}^i} - j \qquad \text{Eq. 26}$$

$$t_j^{n+1} = i + \frac{t_{CFD}^{n+1} - t_{frame}^{i+1}}{t_{frame}^{i+1} - t_{frame}^i} - j \qquad \text{Eq. 27}$$

$$w_{cub}(t_j) = 1.5|t_j|^3 - 2.5|t_j|^2 + 1 \qquad \text{for} 0 \le |t_j| < 1 \qquad \text{Eq. 28}$$

$$w_{cub}(t_j) = -0.5|t_j|^3 - 2.5|t_j|^2 - 4|t_j| - 2 \qquad \text{for} 1 \le |t_j| < 2 \qquad \text{Eq. 29}$$

$$w_{cub}(t_j) = 0 \qquad \text{for} |t_j| \ge 2 \qquad \text{Eq. 30}$$

The interpolated fish boundary coordinates were then calculated using the following:

$$x_{interp}^n = \sum_{j=i-1}^{j=i+2} w_{cub}(t_j^n) \cdot x_{frame}^i \qquad \text{Eq. 31}$$

$$y_{interp}^n = \sum_{j=i-1}^{j=i+2} w_{cub}(t_j^n) \cdot y_{frame}^i \qquad \text{Eq. 32}$$

$$x_{interp}^{n+1} = \sum_{j=i-1}^{j=i+2} w_{cub}(t_j^{n+1}) \cdot x_{frame}^i \qquad \text{Eq. 33}$$

$$y_{interp}^{n+1} = \sum_{j=i-1}^{j=i+2} w_{cub}(t_j^{n+1}) \cdot y_{frame}^i \qquad \text{Eq. 34}$$

After the interpolated data values had been determined, the velocities at the boundary nodes were calculated using Equations 10 and 11. The position was of the nodes was updated using Equations 12 and 13.

The Catmull-Rom spline produced by far the best results out of the three interpolation methods used. This is due to the smoothness that the method provides between FDFs. This smoothness dampens the large spikes of acceleration and fluctuations in the flow field that the other methods were characterized by. For the complete motion UDF with the Catmull-Rom spline interpolation, see Appendix A.

Solution Strategy

This section will cover the various numerical methods that were used in the flow solver. FLUENT offers a wide range of methods to solve for flow fields, and some trial and error was involved in the course of reaching the settings that were eventually used. The FLUENT User Manual (ANSYS Inc., 2009) did offer some guidance on aspects of the solution strategy.

Unsteady Considerations and Numerical Precision

The simulation of a moving body is inherently unsteady. This leads to the problem of determining an appropriate time-step size, number of iterations per time-step, and a reasonable convergence limit. The time-step size was found by setting the CFL condition for mesh deformation close to unity. The CFL condition is defined as:

$$C = \frac{u\,\Delta t}{\Delta x} \qquad \text{Eq. 35}$$

where u is the velocity of the fish boundary, Δt is the time-step size, and Δx is the characteristic cell dimension (Anderson, 1995). The average cell size near the fish and cylinder boundaries was approximately 0.0001-m. The velocity of the fish boundary was highly variable, but never exceeded 1-m/s. For the CFL condition to be equal to one, the solver time-step needed to be set equal to 0.0001-s. The calculation was done using the

upper limit of the velocity of the fish boundary. It was felt that having too small of a time-step would be better than having too large of a time-step.

The convergence limit of the simulation was set to 10^{-5} for all of the tracked quantities. This is a fairly typical convergence limit for this type of simulation. The number of iterations per time-step was set to 50. This was sufficient for the simulation to converge at nearly every time-step. A double precision solver was used because the memory overhead was not very large, and some nodes on the fish boundary moved very little in some time-steps.

Numerical Methods

Discretization

Many of the default methods in FLUENT were used for the current study, as they are applicable for a large range of flow situations. All of the spatial discretization methods used first-order upwind schemes, with the exception of momentum, which used a third-order MUSCL scheme. The MUSCL scheme gave better accuracy during validation, compared to other momentum discretization schemes. The under-relaxation factors were kept at their defaults, as the solver had no issue reaching convergence.

Dynamic Meshing

While the dynamic motion of the fish boundary is controlled in a UDF, there are several settings that must be properly adjusted within FLUENT for dynamic motion and remeshing to occur. The first step was to properly hook the UDF into the FLUENT solver. This was done by activating the dynamic mesh option within FLUENT and associating the dynamic motion UDF with the fish boundary.

There are also settings for smoothing and remeshing in the dynamic mesh module of FLUENT. The smoothing in FLUENT is termed ‘spring based smoothing’. Spring based smoothing moves interior nodes based on the motion of the dynamic boundary.

The amount of motion that the interior nodes undergo is based on the spring constant factor. The spring constant factor is user defined, and can range from zero (no damping) to one (default damping). The effect of the spring constant factor can be seen in Figure13. (ANSYS Inc., 2009). For this study, a spring constant factor of 0.01 was used, as it produced a moderate amount of mesh deformation, and kept the fish motion very close to the FDFs.

The second setting within the dynamic mesh module was 'boundary node relaxation'. This setting determines how much deformation will occur at the boundary during smoothing. This setting also ranges from zero to one, with the value zero preventing any smoothing at the boundary. Because the boundary motion was wholly defined by the UDF, very little boundary smoothing was desired. A value of 0.01 assigned to this option, as it was thought that setting it to zero may cause stability issues. The above smoothing methods are iterative procedures, and were assigned a convergence limit of 10^{-15} with 500 iterations performed.

The final step in setting up the dynamic meshing was to set the criteria for remeshing. FLUENT has a built-in method that calculates reasonable defaults for the minimum and maximum length scales, and skewness. The remesh interval was changed from five to one, however. This was to ensure that the mesh was as high quality as possible throughout the simulations.

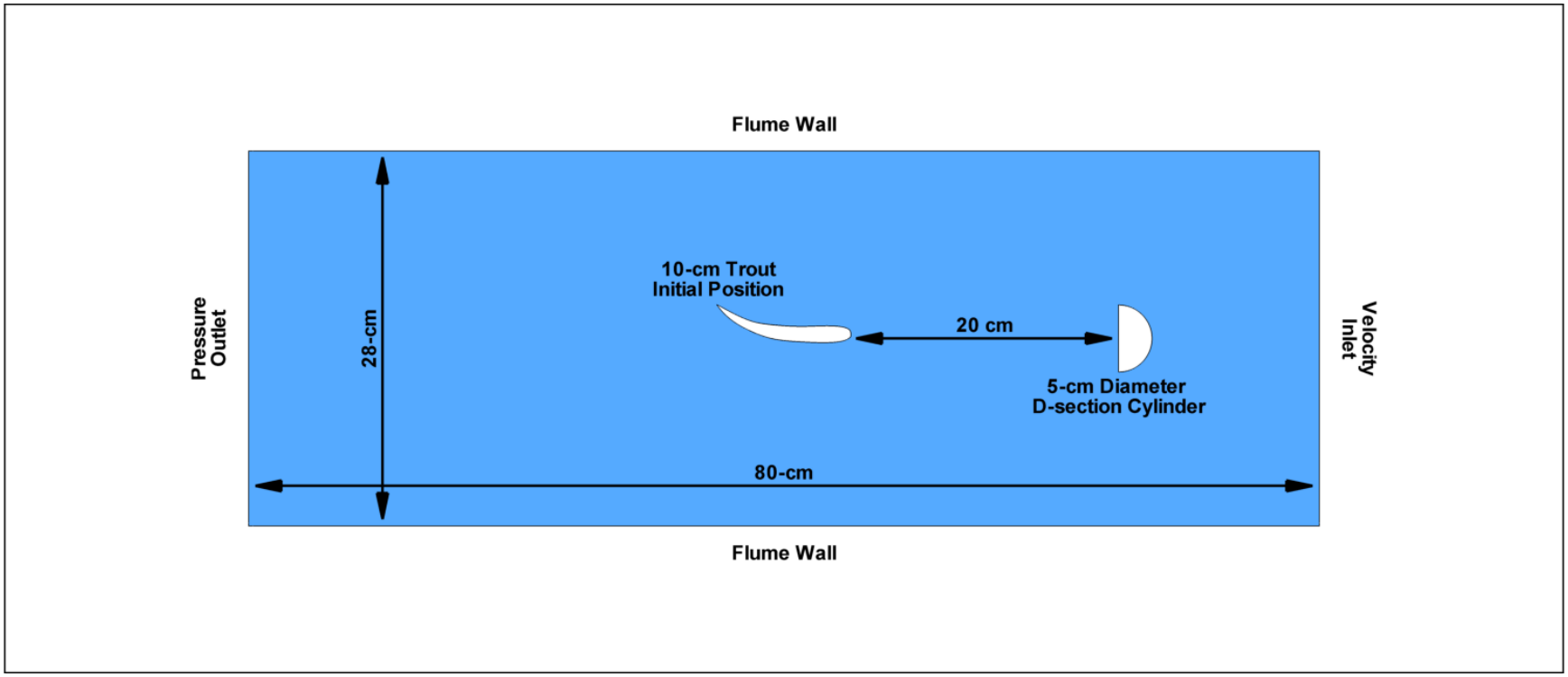


Figure 7: Setup of the simulation. The boundary conditions are discussed in detail within the. The D-section cylinder was not present in all simulations.

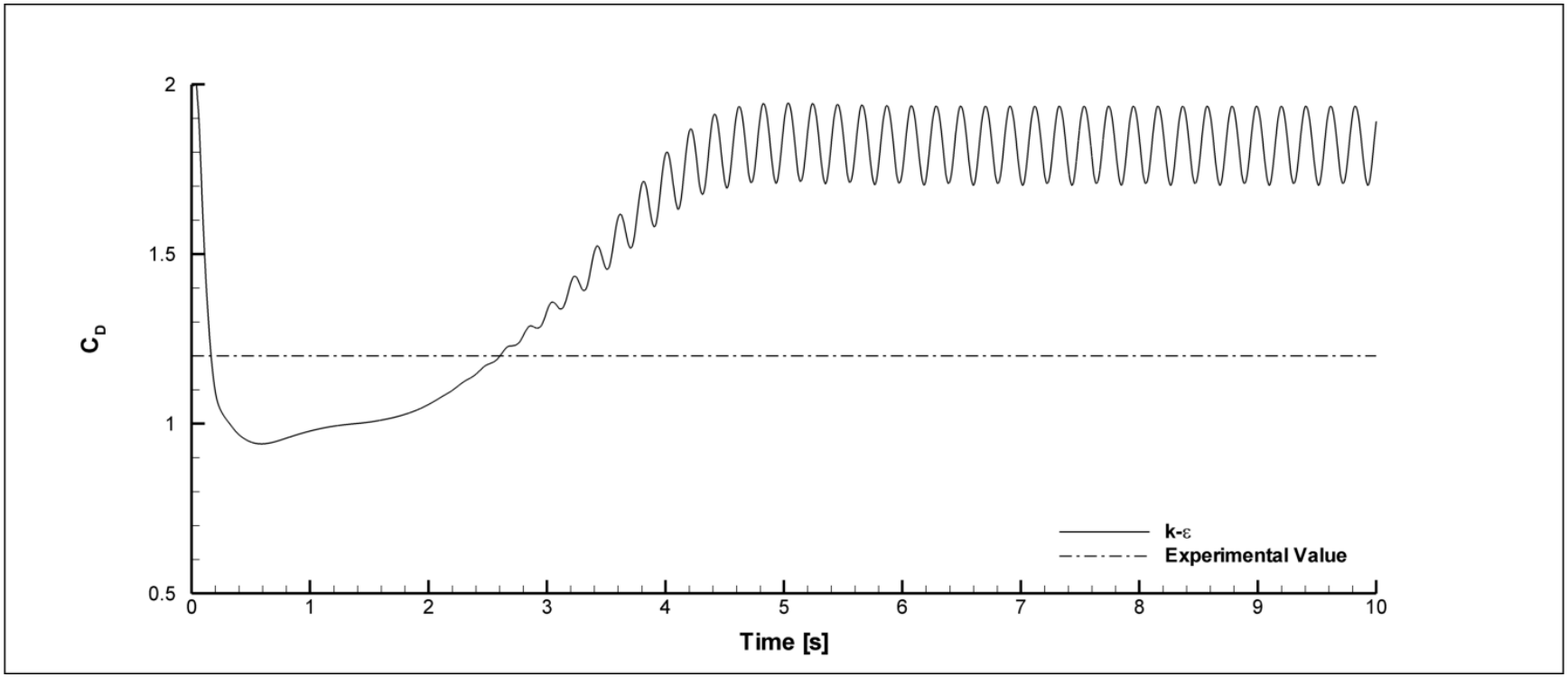


Figure 8: Coefficient of drag on the D-section cylinder using the k-ε turbulence model. This model showed very regular vortex shedding. This model over predicted experimental drag estimates. According to Benim *et al*. (2008), this over prediction is to be expected.

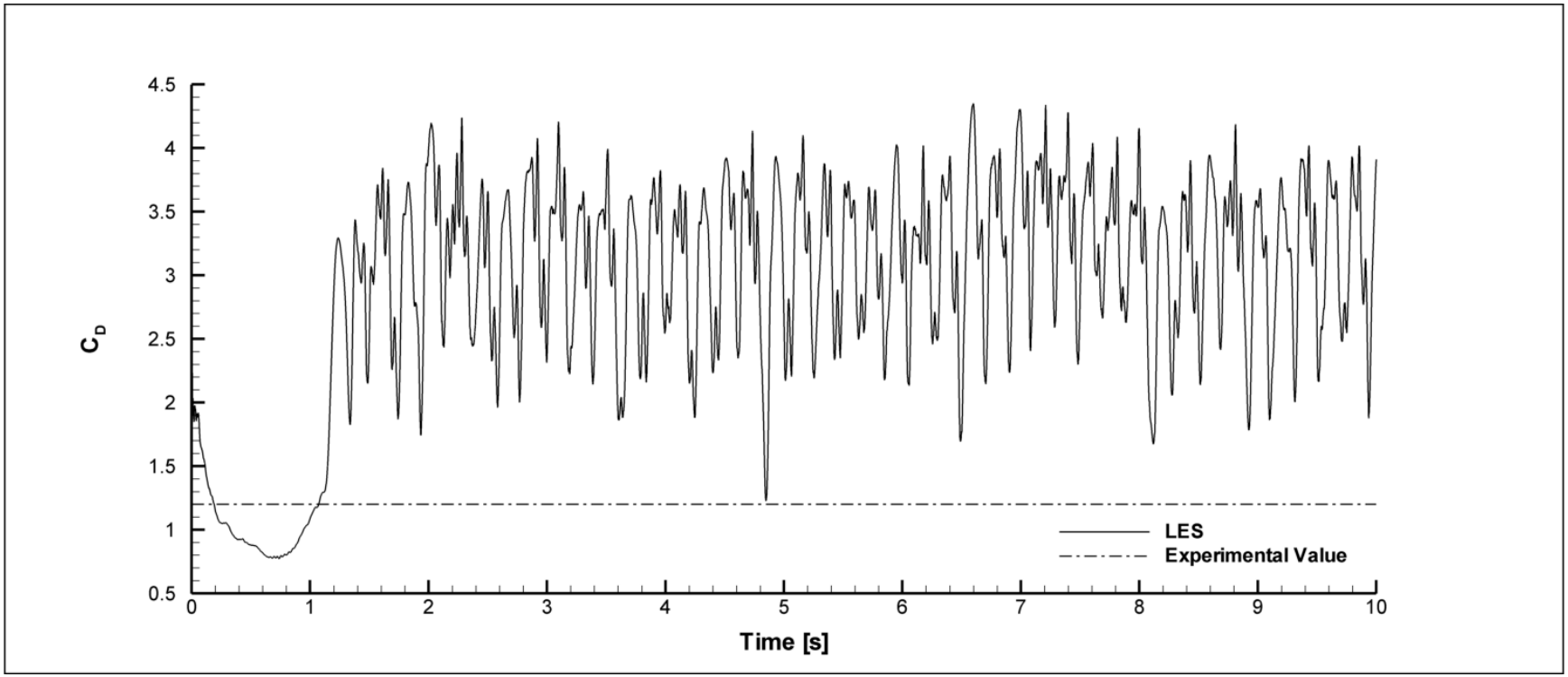


Figure 9: Coefficient of drag on the D-section cylinder using the LES turbulence model. This model showed very poor agreement with the experimental data.

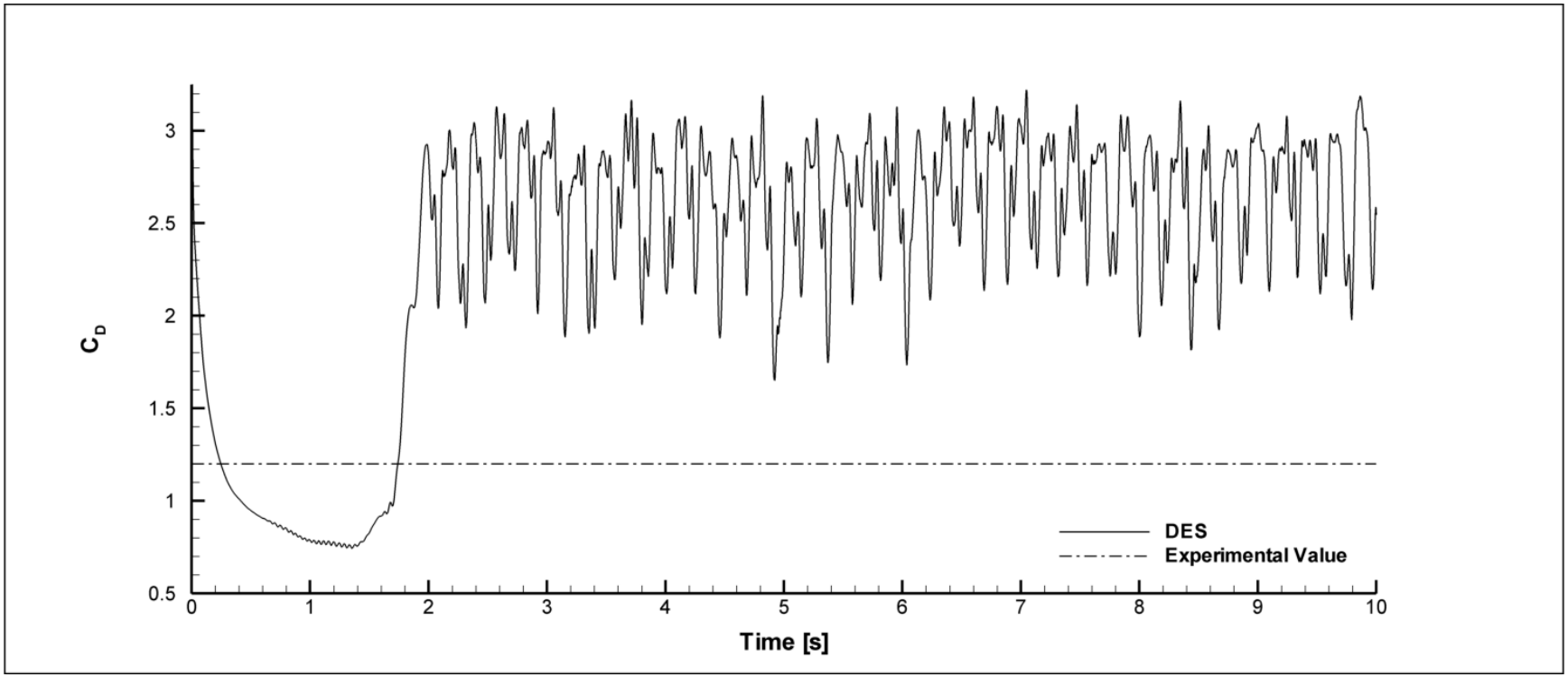


Figure 10: Coefficient of drag on the D-section cylinder using the DES turbulence model. This model's performance fell between that of the k-e model and LES

Table 1: Number of nodes and cells in each simulation configuration

		Number of Nodes					
Kinematics	**Configuration**	**Flume Walls**	**Inlet/Outlet**	**Fish Body**	**D-Section Cylinder**	**Total in Domain**	**Cell Count**
Free Stream	No-Cylinder	250	125	1569	N/A	56,471	109,721
Free Stream	Cylinder	250	125	1569	911	45,067	87,820
Karman Gait	No-Cylinder	250	125	1561	N/A	45,168	88,019
Karman Gait	Cylinder	250	125	1561	930	56,853	110,463

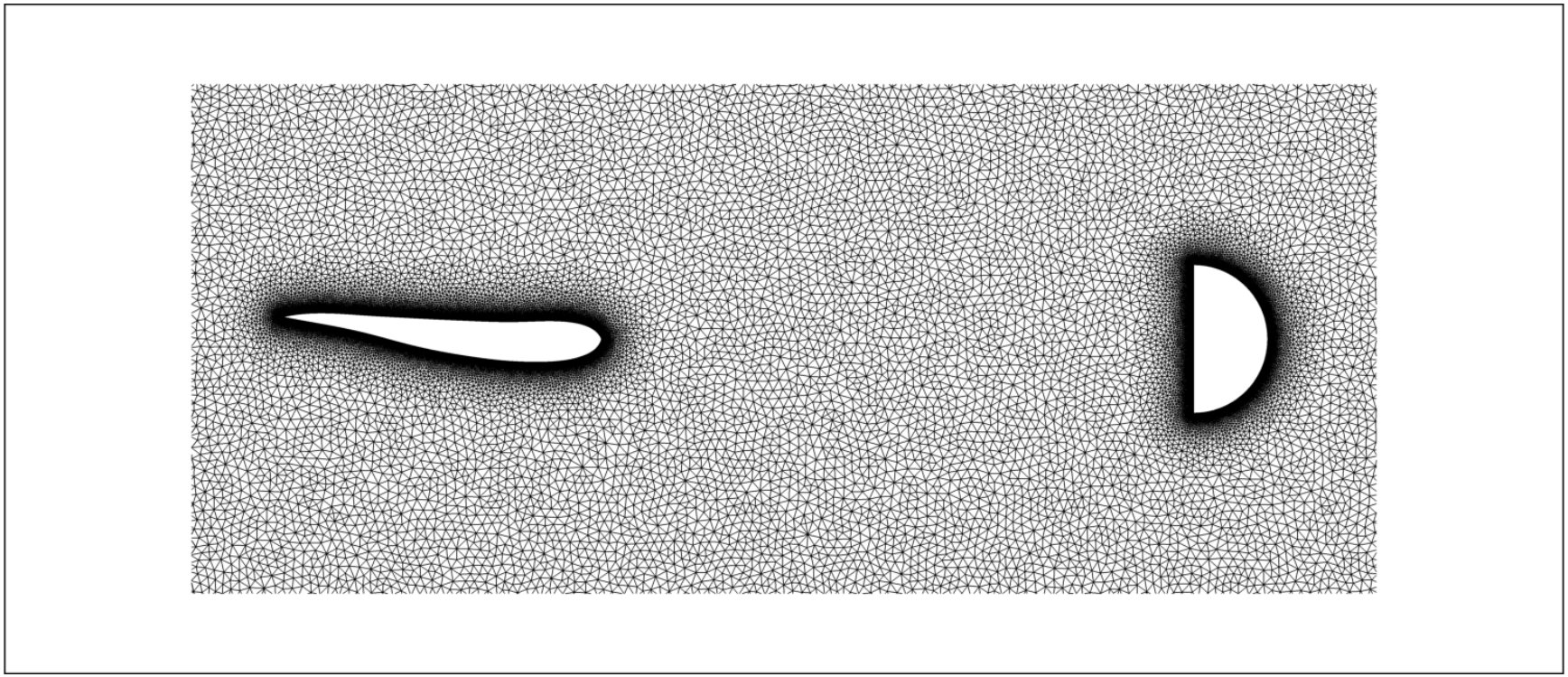


Figure 11: Close-up view of the initial position of the free-stream fish with the D-section cylinder present. The grid with the D-section cylinder removed is quite similar to this one, except for a lack of refinement where the D-section cylinder is located.

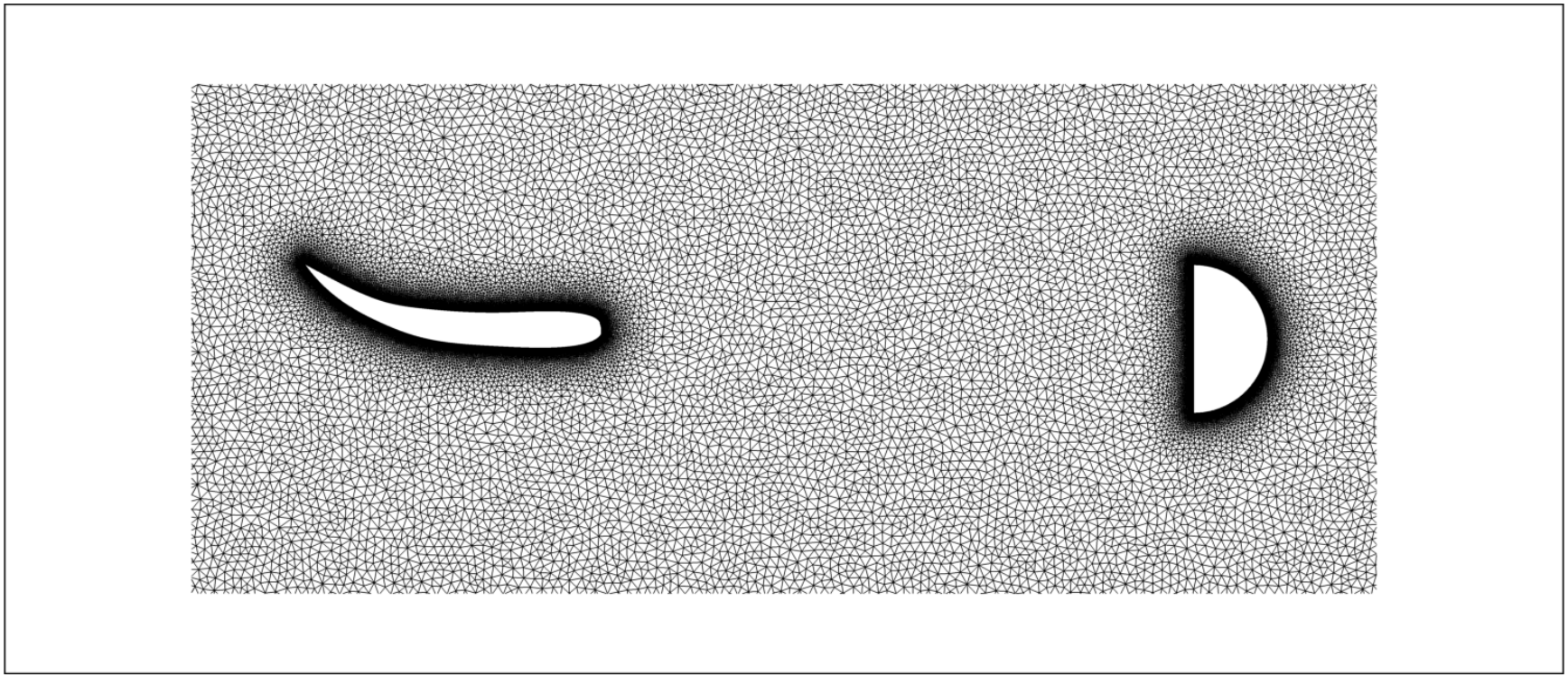


Figure 12: Close-up view of the initial position of the Kármán gait fish with the D-section cylinder present. The grid with the D-section cylinder removed is quite similar to this one, except for a lack of refinement where the D-section cylinder is located.

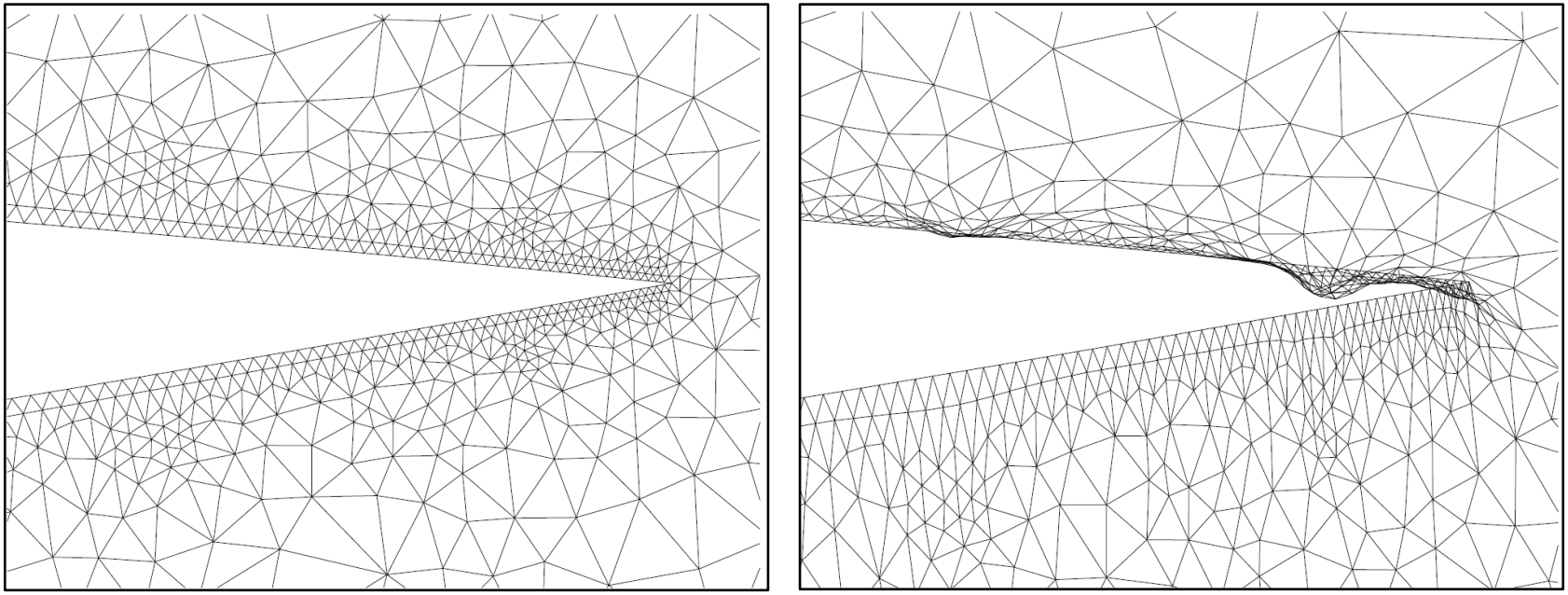


Figure 13: Comparison of a spring constant factor of zero (left) and one (right). It can be seen that excess deformations occur when the spring constant is set to a large value. The deformations shown on the right look like they would lead to instabilities in the solver.

Source: ANSYS Inc., 2009

CHAPTER IV

RESULTS AND DISCUSSION

Introduction

Eleven simulations were completed in the course of this study. The first simulation completed was a validation test case. In this simulation, the unsteady flow field around a D-section cylinder was solved for. This simulation was designed to test the accuracy of the numerical model, and ensure that it was suitable for use to simulate fish motion.

The remaining ten simulations solved for the flow field around swimming fish, with two flume configurations, and two types of swimming kinematics. Eight simulations were completed with both the fish and D-section cylinder being present in the flume, and two simulations were completed with only the fish in the flume.

Validation of the Numerical Model

Validation of the numerical model was accomplished by solving for the flow field around a D-section cylinder and comparing the numerical results with experimental work. Parameters that were validated were the frequency of vortex shedding and the coefficient of drag on the cylinder. Validation was not possible using fish within the flume, because there is no available data to validate against.

Expected Results

Vortex Shedding Frequency

The frequency (and wavelength) of vortex shedding is a function of Strouhal number. The Reynolds number in validation simulation was calculated as follows:

$$Re = \frac{U_f d}{\upsilon} = \frac{0.45 \cdot 0.05}{1.156 \cdot 10^{-6}} \approx 20{,}000 \qquad \text{Eq. 36}$$

whereU_f is the free-stream velocity, d is cylinder diameter, and ν is the kinematic viscosity. For this flow regime, the Strouhal number is 0.2. To calculate the expected vortex shedding frequency, the following equation can be used (Liao, 2003):

$$St = \frac{fd}{U} \quad \text{Eq. 37}$$

where St is the Strouhal number, f is the frequency of vortex shedding, and U is the velocity in the vicinity of the cylinder. The velocity near the cylinder is higher than the free-stream velocity, U_f, due to flow constriction. The constricted velocity is calculated as follows:

$$U = U_f \times \frac{W}{W-d} \quad \text{Eq. 38}$$

where w is the width of the flume. The equation to convert the vortex shedding frequency to vortex wavelength, λ, is given as

$$\lambda = \frac{U_f}{f} \quad \text{Eq. 39}$$

After some rearrangement, the three above equations become

$$\lambda = \frac{d(w-d)}{St \cdot w} \quad \text{Eq. 40}$$

which gives a vortex wavelength of 20.5-cm.

Drag Coefficient

Experimental studies have shown that the coefficient of drag on a D-section cylinder is equal to 1.2 (White, 2008). However, Benim *et al*. (2008) have shown that 2D unsteady turbulence models over-predict the drag coefficient on cylinders (by 45.22 percent in their case). It is hypothesized by Benim *et al*. that the discrepancy is due to three dimensional effects that are neglected in 2D simulations. Therefore, it would seem

reasonable to expect a two dimensional drag coefficient to fall somewhere between 1.2 and 1.8.

Numerical Results

Vortex Wavelength

Vortex shedding was observed in the flow field behind the D-section cylinder. This showed that the selected turbulence model was capable of resolving flow separation. Figure 14 shows the z-vorticity in the flow field. The wavelength of vortex shedding was found to be 16.6-cm. This is an under prediction of approximately 19%. The under prediction of the vortex shedding wavelength was of some concern. This parameter was vital, since the simulated fish using Kármán gait kinematics was in reality swimming between vortices separated by 20.5-cm.

Drag Coefficient

The drag coefficient was calculated using built-in FLUENT functions. An image of the drag coefficient history on the D-section cylinder can be found in Figure 8 (Chapter III). The time-averaged drag coefficient was found to be 1.824 in the validation case. This is a deviation of 52% from the experimental value. However, when compared to the expected upper limit in two-dimension numerical studies, it is a deviation of just 1.33%. While it is unfortunate that the drag coefficient does not reflect experimental results, it seems that the model is performing fairly well, considering the two-dimensionality of the simulations. In addition, this study was focused on the relative efficiency differences between different kinematic and flume configurations, and less on the absolute values of efficiencies.

Summary

The validation of the numerical model was mixed. The wavelength of vortex shedding was under predicted by what seemed like a significant amount. The impact of

this under prediction cannot truly be known, but qualitatively, it appeared to impact the results a small amount. The over prediction of drag on the D-section cylinder was to be expected, as stated by Benim *et al.* (2008). Using a more robust turbulence model (such as DES or LES) and moving the simulations to the third dimension will most likely alleviate these problems.

Predicted Flow Fields and Simulation Flow Fields

Introduction

This series of ten simulations examined the flow field conditions and propulsive efficiencies produced by four distinct configurations of fish swimming kinematics and flume geometry. Each simulation was first run to steady-state before the fish motion was initiated. The simulations were confirmed to have reached steady-state by examining the drag history on the fish and D-section cylinder (if present). The initialization of the flow field was done to remove initial transient effects that may have influenced results.

The configurations that contained a D-section cylinder showed an oscillatory wake. This was from the production of a Kármán vortex street. This presented a challenge in determining when the motion of the fish was to be initiated. There were two obvious choices: 1) run the simulation to steady-state and perform a qualitative visual inspection and 2) examine the drag history on the stationary fish body and initiate motion when the drag reaches local minima and maxima. The second option was selected. Figures 15 and 16 show the drag history on the fish using free-stream kinematics and Kármán gait kinematics, respectively. Visual inspection of the flow field after the simulations were completed showed that this method worked fairly well.

Predicted Flow Field

According to the hypothesis, the most efficient simulation test case should be the combination of Kármán gait kinematics with a flume containing a D-section cylinder.

This result has been shown to be the case by Liao (2004), Taguchi and Liao (2011), and Cook and Coughlin (2010). The second highest efficiency was then predicted to be the combination of free-stream kinematics and an empty test flume. Finally, the lowest efficiencies were expected to be in the remaining two configurations (free-stream kinematics with a D-section cylinder and Kármán gait kinematics with an empty flume). It was unknown which of these two cases would be the more efficient.

Extrapolation from the phase diagram in Schnipper *et al*. (2009) (Figure 17) predicts that 2S inverse Kármán streets will be formed in the wake of both types of locomotion. The data used in this prediction can be found in Table 2. These wakes are thrust producing, characterized by two vortices of alternate sign being shed by the moving fish during every period of motion.

Flow Field with Free Stream Kinematics

The data that was extracted during image analysis was able simulate approximately four cycles of free-stream swimming, over a total simulation time of 0.572 seconds.

Flume with No Cylinder

Free-stream kinematics paired with an empty flume is one of the two experimentally observed configurations tested. Figure 18 shows a series of images showing the development of the flow field as the simulation progresses. Figure 18 shows the swimming fish producing a 2S inverse Kármán street wake, as predicted. This configuration was hypothesized to be the second most efficient configuration, and the results from the flow field looked as expected.

Flume with Cylinder

Free-stream kinematics paired with a flume containing a D-section cylinder was a configuration that had no physical counterpart. The purpose of this configuration was to

determine the performance of free-stream swimming within a Kármán street, to that of the Kármán gait within a Kármán street. There were four simulations completed using this configuration. The motion was initiated at different times, according to the oscillatory state of the Kármán street caused by the D-section cylinder.

Figure 19 shows the development of the wake for one of the simulations (all four simulations showed similar results). All of the simulations showed the fish swim directly into the vortex street. The positively rotating vortices can be seen moving across the top of the fish's body, while the negatively rotating vortices move across the bottom of the fish's body. This causes the fish to be swimming through vortices that are moving fluid in the upstream direction, increasing drag on the fish's body.

The wake behind the fish was difficult to discern. The fish appeared to be shedding vortices in an inverse Kármán street, but interference from the Kármán street from the cylinder made it difficult to determine. The vorticity in the wake was much more dissipative in this case compared to the empty flume simulation. It appeared that the vortices from the D-section cylinder and the vortices shed by the fish were cancelling each other out. This is producing a wake that has a limited amount of vorticity at the far downstream end of the flume.

Flow Field with Kármán Gait Kinematics

The data that was extracted during image analysis was able simulate approximately two cycles of Kármán gait swimming, over a total simulation time of 0.732 seconds.

Kármán Gait Kinematics in Flume with No Cylinder

Kármán gait kinematics paired with an empty flume is the other simulation configuration that has no physical counterpart. The Kármán gait swimming mode seems makes little sense in an empty flume. The wide lateral motions that the fish undergoes

are an unnecessary expenditure of energy, which is why it was believed that this case would perform poorly compared to the physically occurring configurations.

Figure 20 shows the development of the flow field in this configuration. A defined inverse Kármán street can be observed, with two vortices of opposite sign being shed during every cycle of motion. This wake structure was predicted using Figure 17 and Table 2.

Kármán Gait Kinematics in Flume with Cylinder

Kármán gait kinematics paired with a flume containing a D-section cylinder was the final configuration tested, and the second physically observed configuration. According to previous studies, the propulsive efficiency in this configuration should be the highest tested. There were four simulations completed in this configuration, one for each of the local minima and maxima. Of the four simulations, only one was successful in simulating the fish slaloming between the vortices shed by the D-section cylinder – which is what is seen in experiments. An example of the flow field development during the three cases in which the slaloming motion was incorrect can be seen in Figure 21. In these three cases, the fish swam directly through the vortices shed by the D-section cylinder. The wake from the fish seemed to amplify the magnitude of the vorticity in the Kármán street. Instead of creating a thrust producing wake, the fish created a stronger drag wake than found with the D-section cylinder alone.

Flow field development for the best performing simulation in this configuration can be seen in Figure 22. Here it can be seen that the fish slaloms correctly. The fish oscillates between the vortices shed by the D-section cylinder, just barely colliding with them. The fish appears to be redirecting the vortices in to an inverse Kármán street. This implies that the fish is capturing energy from the wake of the D-section cylinder, and increasing its propulsive efficiency. While this is the best simulation for this configuration, it seems like there is some undesired interference occurring between the

fish and the wake of the D-section cylinder. This may be caused by several factors. One possible explanation is that the motion of the fish is not being initiated at the exactly correct flow time. Another explanation is that the wavelength of the vortices shed by the D-section cylinder is slightly lower than the experiments have predicted, as seen in the validation case above. Experimental studies predicted a vortex wavelength of approximately 20.5-cm, while the numerical results produced a vortex wavelength of only 16.6-cm. A vortex shedding wavelength under prediction of 19% may be adversely affecting the results.

Propulsive Efficiencies

Data Reporting UDF

The propulsive efficiency for each swimming fish was calculated using a method adapted from Borazjani and Sotiropoulos (2008, 2010). Borazjani and Sotiropoulos used this method to make similar calculations on three-dimensional swimming fish. The following equations were used to determine propulsive efficiency.

$$F(t) = \int_A \left[-pn_x + \tau_{xx}n_x + \tau_{xy}n_y\right] \mathrm{d}A \qquad \text{Eq. 41}$$

Equation 41 calculates the total force applied to the fish's body over time, where p is pressure, n_j is the unit normal vector in the jth direction, and τ_{ij} is the viscous stress tensor. The total force is then decomposed into thrust and drag components in Equations 42 and 43.

$$T(t) = \frac{1}{2}\left[\int_A -p_x n_x \mathrm{d}A + \left|\int_A p_x n_x \mathrm{d}A\right|\right] + \frac{1}{2}\left[\int_A \left(\tau_{xx}n_x + \tau_{xy}n_y\right)\mathrm{d}A + \left|\int_A \left(\tau_{xx}n_x + \tau_{xy}n_y\right)\mathrm{d}A\right|\right] \qquad \text{Eq. 42}$$

$$-D(t) = \frac{1}{2}\left[\int_A -p_x n_x \mathrm{d}A - \left|\int_A p_x n_x \mathrm{d}A\right|\right] + \frac{1}{2}\left[\int_A \left(\tau_{xx}n_x + \tau_{xy}n_y\right)\mathrm{d}A - \left|\int_A \left(\tau_{xx}n_x + \tau_{xy}n_y\right)\mathrm{d}A\right|\right] \qquad \text{Eq. 43}$$

The lateral power output is calculated in Equation 44, where $\dot{h}$ is the lateral velocity of the fish boundary.

$$P_L = \int -pn_y\dot{h}_y\,dA + \int(\tau_{yx}n_x\dot{h}_y + \tau_{yy}n_y\dot{h}_y)\mathrm{d}A \qquad \text{Eq. 44}$$

The propulsive efficiency is finally calculated using Equation 45:

$$\eta = \bar{T}U/(\bar{T}U + \bar{P}_L) \qquad \text{Eq. 45}$$

where the over bars denote time-averaged quantities, and *U* is the free-stream velocity.

Equations 41, 42, 43, 44 were implemented using a FLUENT UDF. Equation 45 was calculated after the simulations had completed using Microsoft Excel. This was necessary due to the time-averaging required. The source code for the data reporting UDF in its entirety can be found in Appendix B. This code works for both serial and parallel FLUENT solvers, and should work for all bodies undergoing motion in both 2D and 3D, with very minor modifications.

Results

Force, thrust, drag, and power data exported by the data reporting UDF was filtered to remove erroneous data. There were occasional spikes in the outputted data that were obviously incorrect. When an outlier was encountered, none of the data was used from that specific time-step. The time-averaging of the data only included full swimming cycles, no partial cycles were included in the analysis.

Table 3 contains the time-averaged variables from all of the simulations. Free-stream kinematics outperform Kármán gait kinematics in the simulations in which no D-section cylinder was present. Free-stream kinematics are approximately 7% more efficient in this configuration. This is because the Kármán gait kinematics required almost double the power of the free-stream kinematics, while only producing one and a

half times as much thrust. The increased power requirement is due to the large lateral displacements used in the Kármán gait.

There is little variation in the results of the simulations in which free-stream kinematics are paired with a flume containing a D-section cylinder. The thrust production in these simulations can be seen to be significantly lower than the thrust production of the simulation in which free-stream kinematics are paired with an empty flume. This is most likely due to interaction between the fish and the Kármán vortex street created by the D-section cylinder.

When the Kármán gait kinematics are paired with a flume containing a D-section cylinder, the results are highly variable. The three lowest efficiencies (23.0%, 29.8%, and 31.0%) were found in this configuration, as well as the highest efficiency (50.1%). The simulations with the very low efficiencies did not model Kármán gait swimming accurately. In these simulations, the fish collided with the vortices shed by the D-section cylinder, increasing power output, and reducing swimming efficiency. In the high efficiency case, the fish slaloms in a way that looks similar to what is observed experimentally. However, as stated previously, it does seem that the slaloming of the fish could be improved, either by initiating the motion in a more accurate way, or by using a turbulence model that produces a vortex shedding wavelength more similar to that observed in experiments.

Discussion

The results from the ten simulations have reinforced the hypothesis that Kármán gait kinematics in the presence of a Kármán street is the most efficient mode of swimming, followed by free-stream kinematics in an empty flume which has a constant undisturbed upstream velocity field. Kármán gait kinematics in an empty flume was shown to be more efficient than free-stream kinematics in the presence of a Kármán street. The simulations that modeled completely unphysical pairings of flow fields and

kinematics (Kármán gait with no cylinder and free-stream kinematics with a cylinder) performed significantly worse than the physical pairings of flow fields and kinematics. This reinforces both the hypothesis and our numerical methodology.

It is proposed that the increased efficiency of Kármán gait kinematics in the presence of a Kármán vortex street is due to energy extraction from the vortex street. This seems reasonable, as the fish was shown to drastically alter the wake structure, changing it from a drag wake, into a thrust producing wake. This explains the slaloming motion that the fish undergoes during the Kármán gait.

However, the difference in efficiency between the two physical configurations was not great. It was expected that the efficiency gains of using the Kármán gait in the presence of a Kármán street would be greater than a couple of percent. To get a truly accurate picture of what is occurring, more work needs to be done. At a minimum, the simulations should take place in three dimensions, and use a better performing turbulence model.

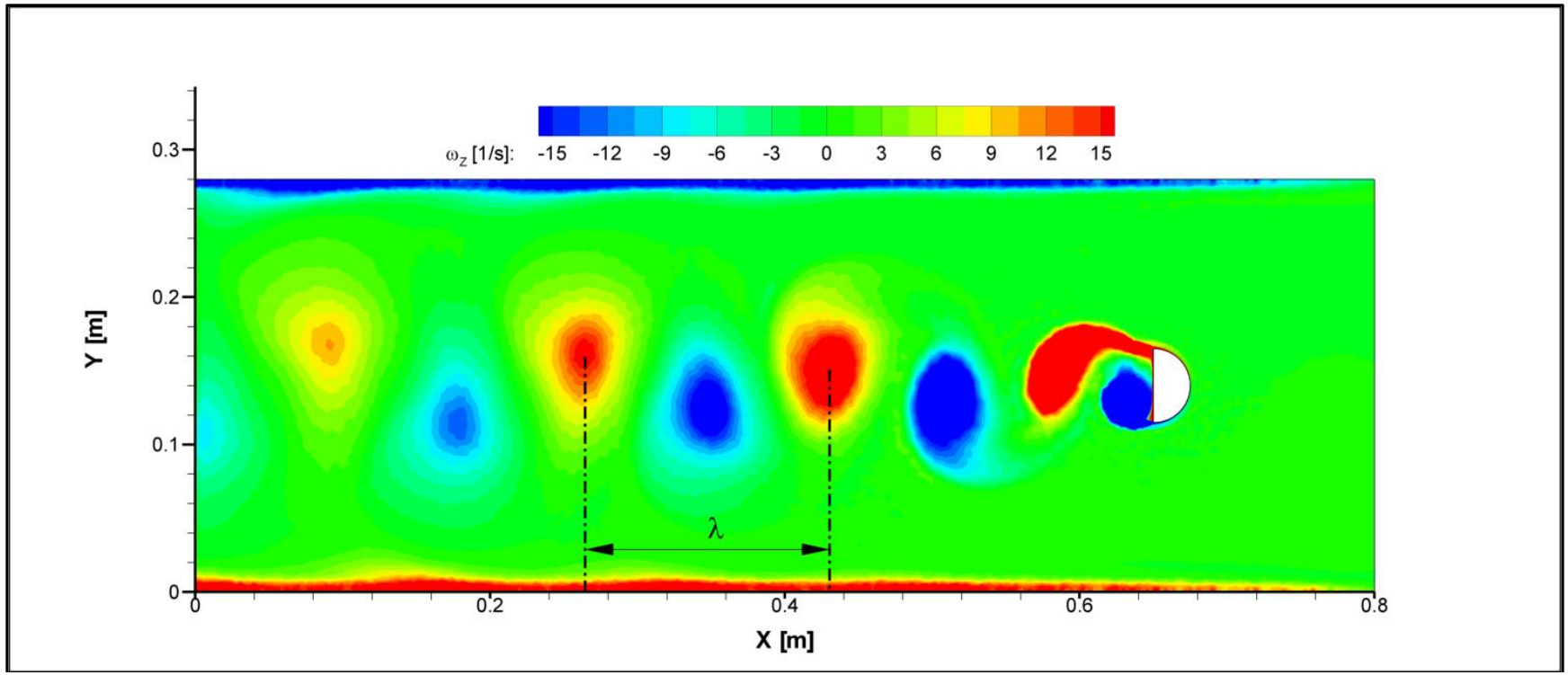


Figure 14: Test case showing the validation of the vortex shedding wavelength, which was found to be 16.5-cm.

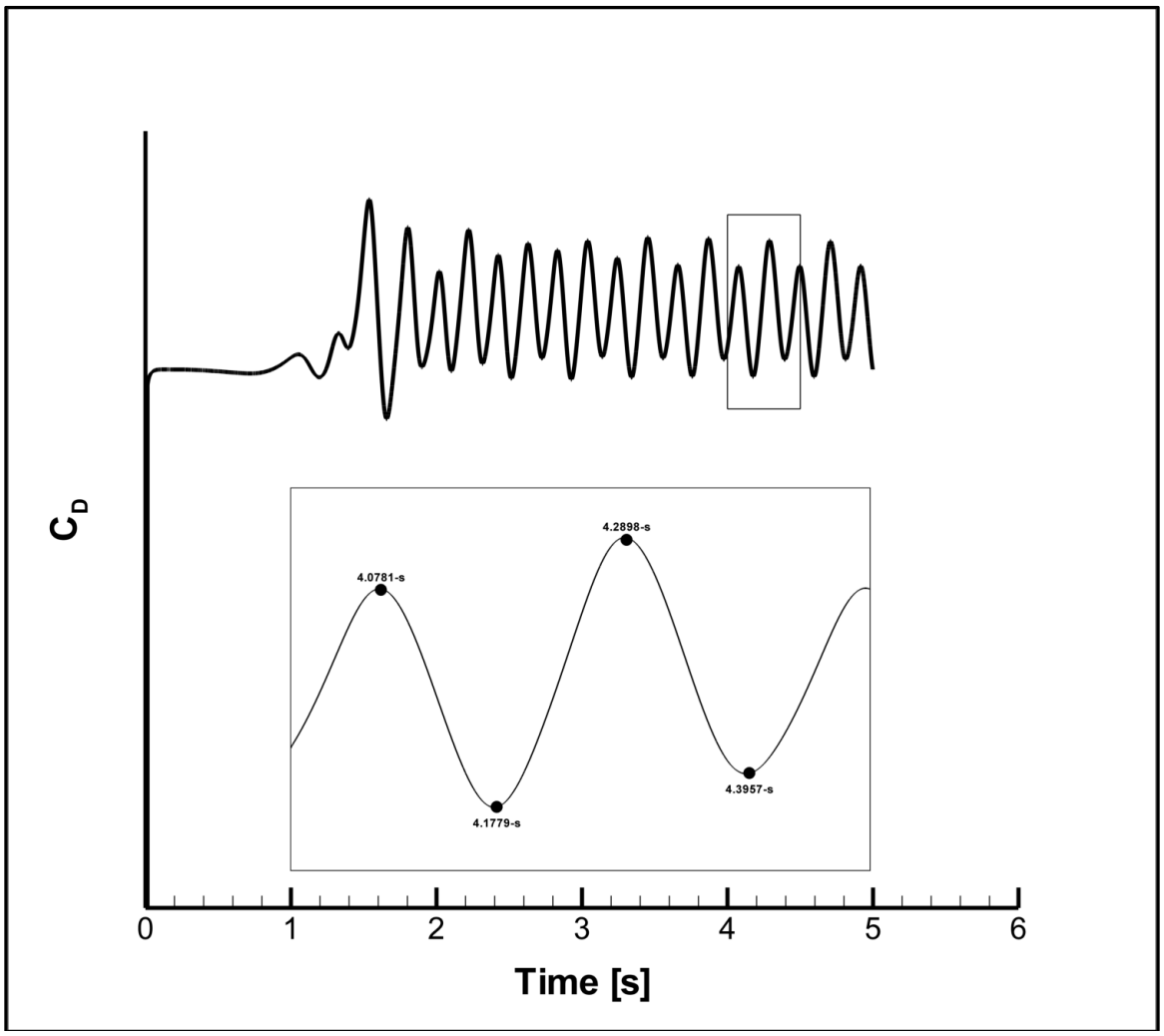


Figure 15: Drag history on the free-stream fish in the presence of a Kármán vortex street. Motion of the fish was initiated at the local maxima and minima shown.

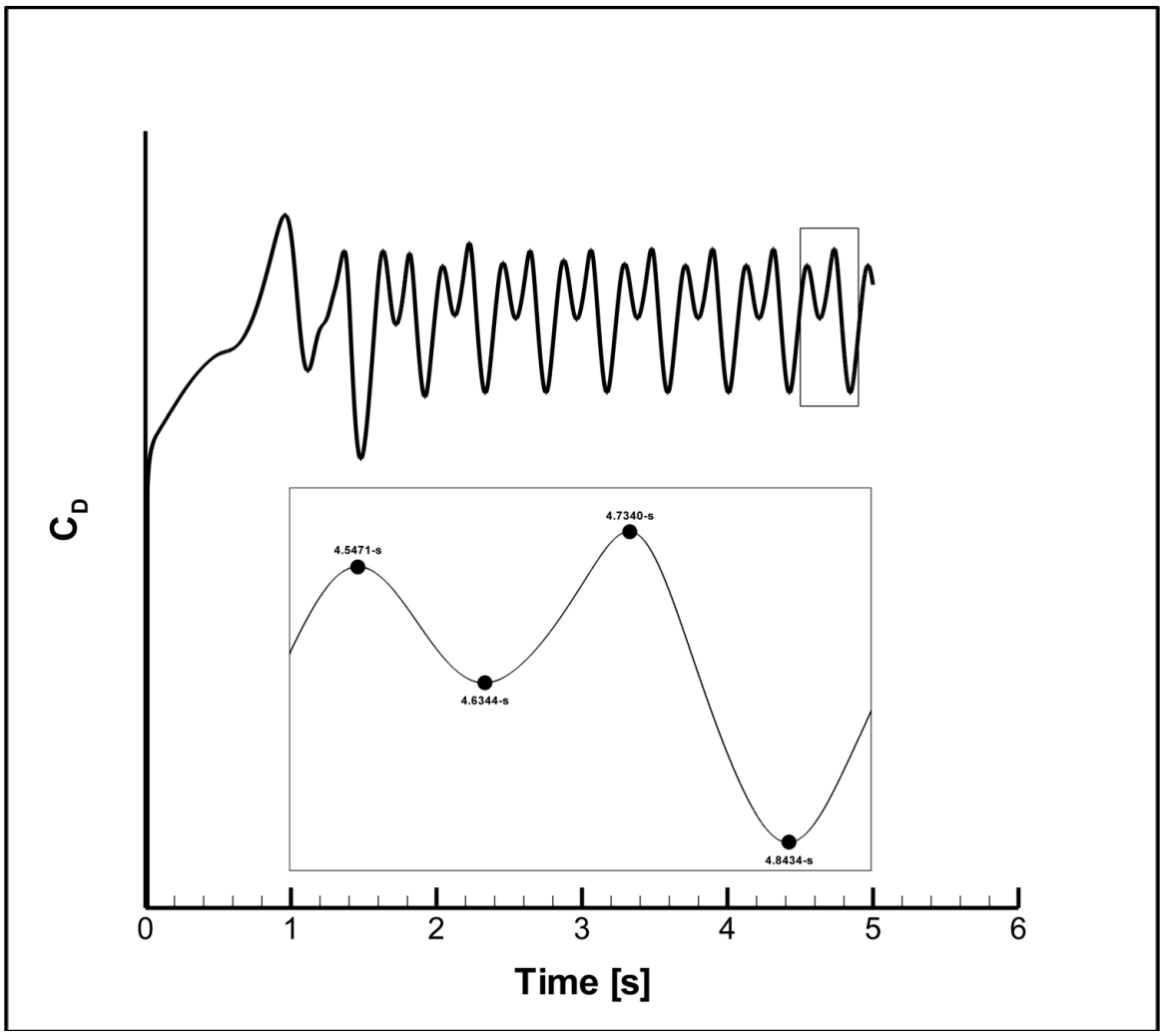


Figure 16: Drag history on the Kármán gait fish in the presence of a Kármán vortex street. Motion of the fish was initiated at the local maxima and minima shown.

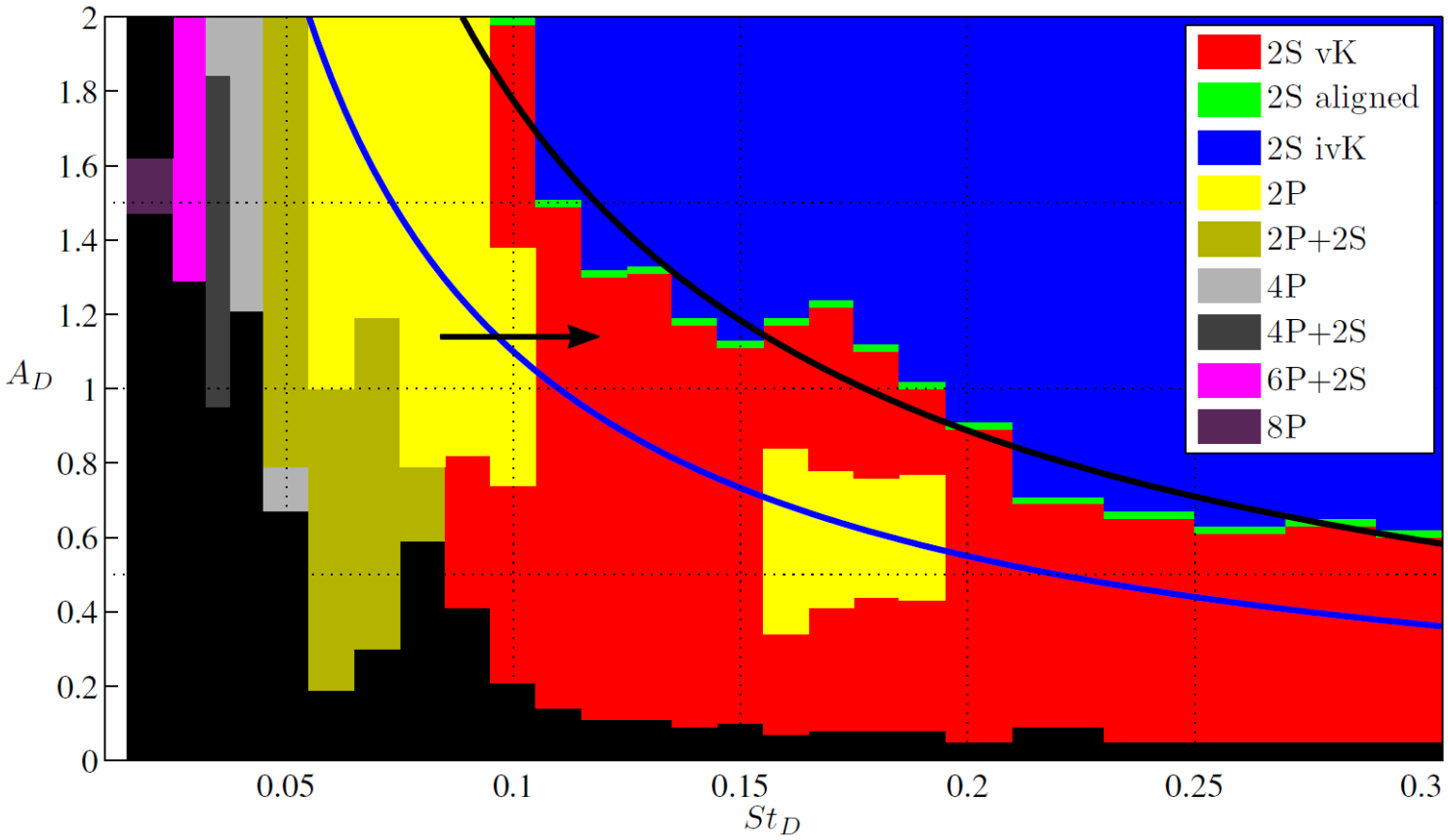


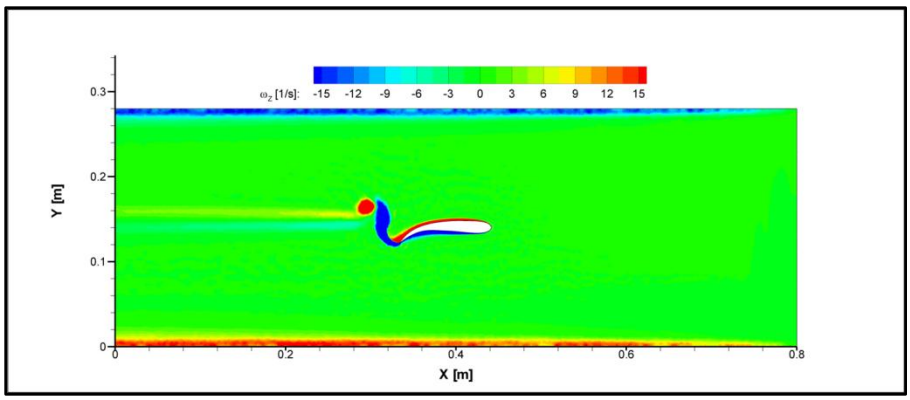
Figure 17: Phase diagram showing the expected wake pattern based on Strouhal number and dimensionless amplitude.

Source: Schnipper *et al.*, 2009

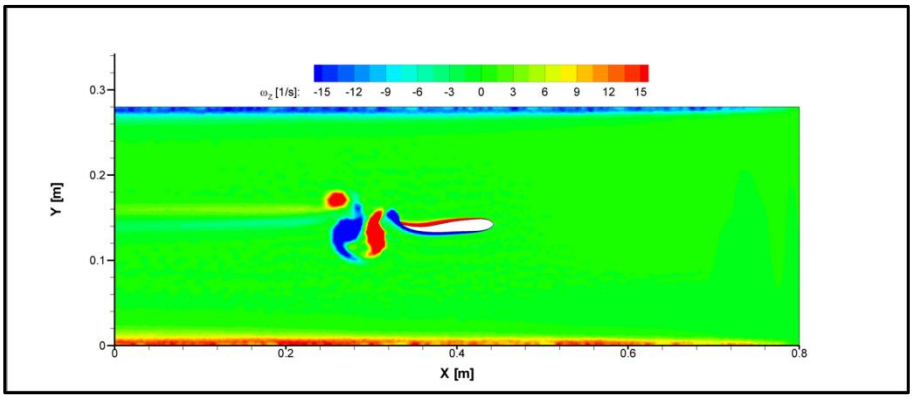
Table 2: Simulation parameters used to determine the predicted wake structure.

Kinematics	Amplitude, A [m]	Frequency, f [Hz]	Length Scale, D [m]	A_D [$2A/D$]	St [Df/U]
Free-stream	0.016	6.993	0.014	2.380	0.213
Kármán gait	0.038	2.732	0.012	6.090	0.076

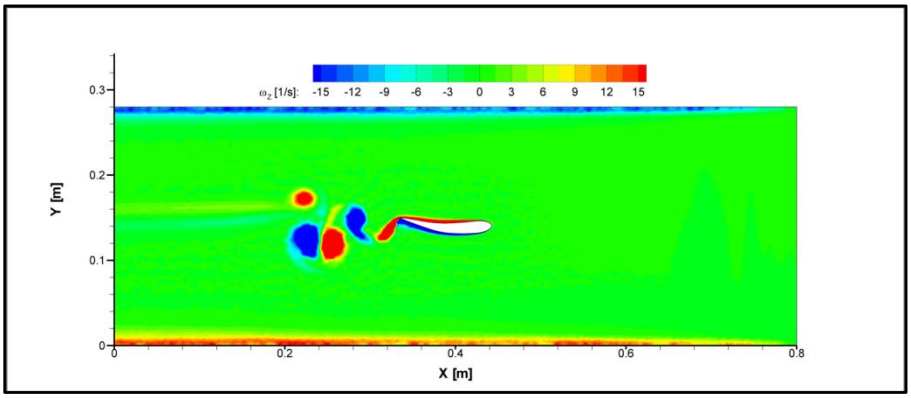
Note: The length scale, *D*, used in these calculations is the width of the fish, not the length of the fish. This is in accorance to how Schnipper *et al*. (2009) performed their calculations.



(A)

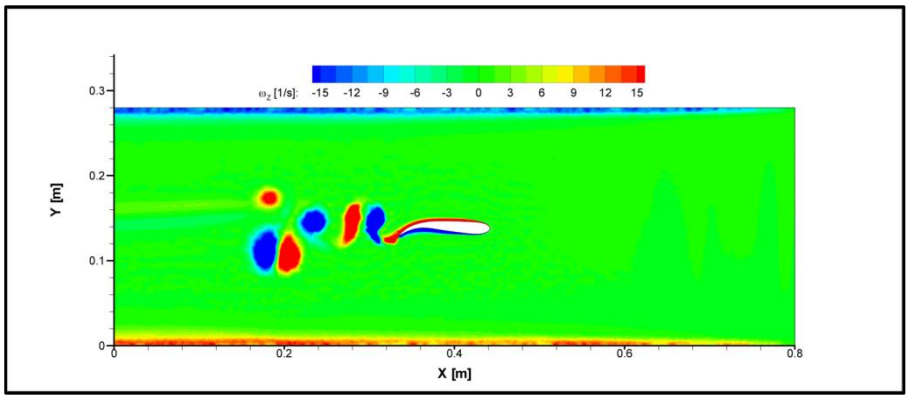


(B)

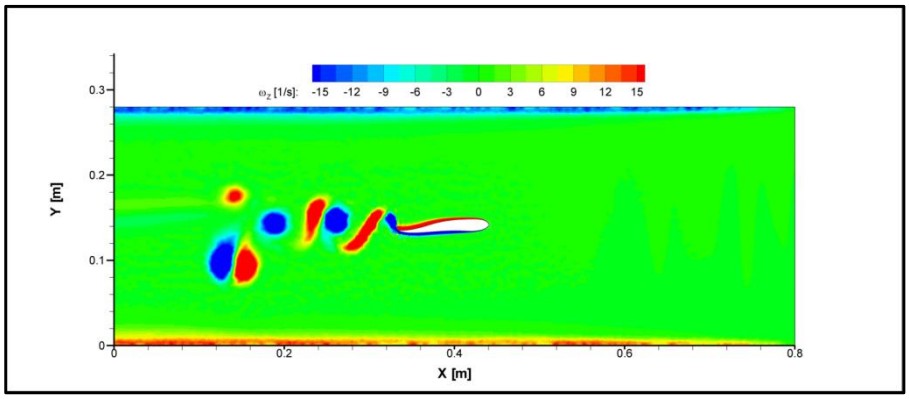


(C)

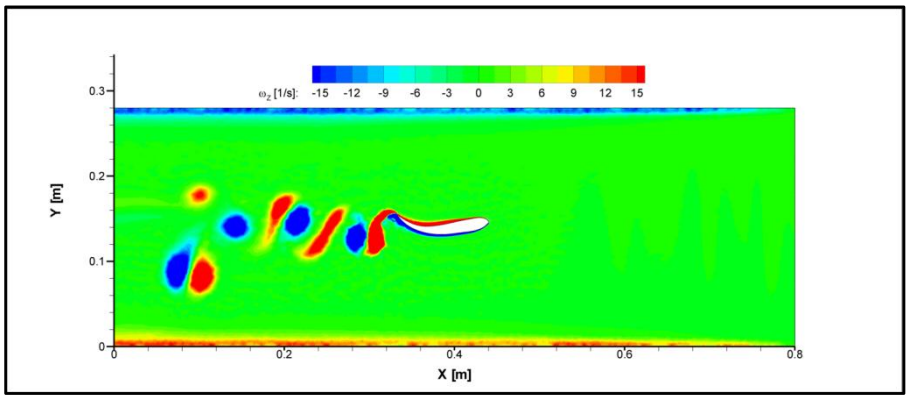
Figure 18: Flow field evolution of free-stream kinematics paired with an empty flume.



(D)

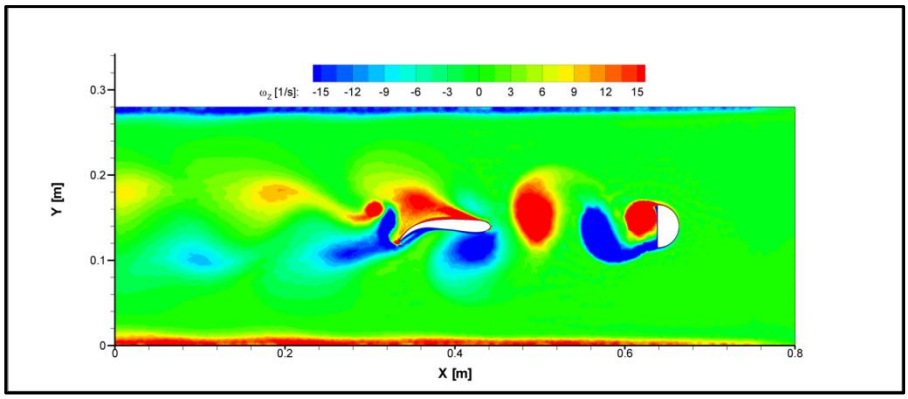


(E)

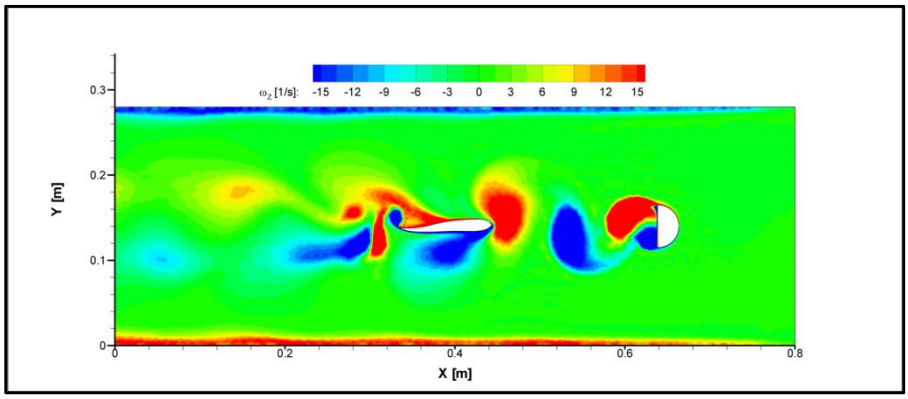


(F)

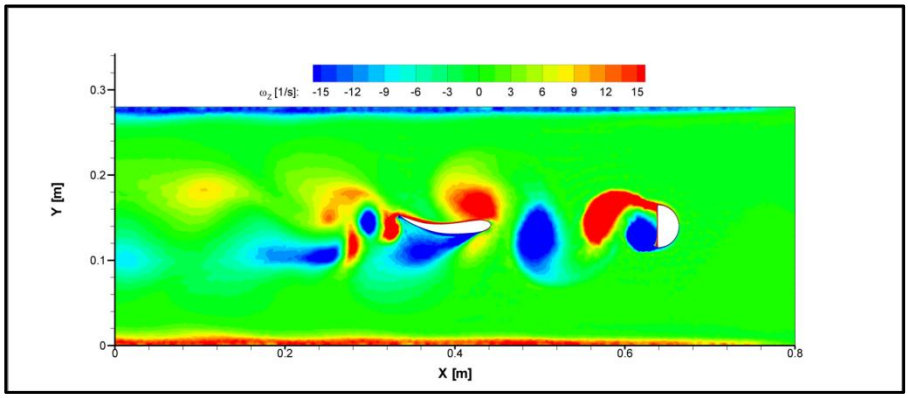
Figure 18: Continued.



(A)

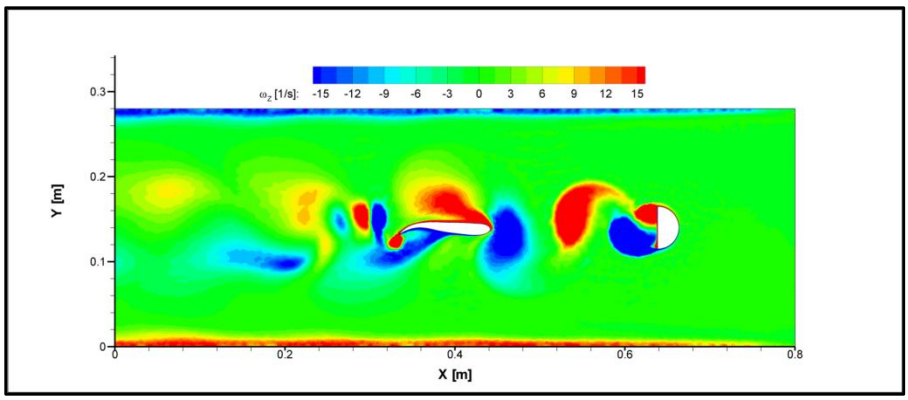


(B)

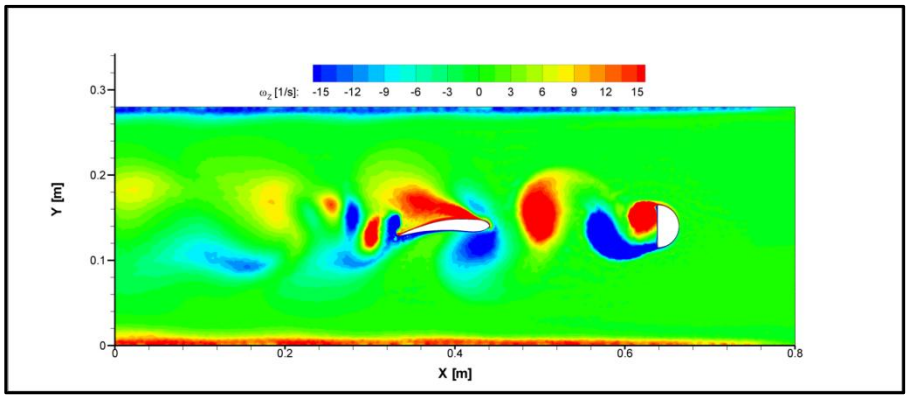


(C)

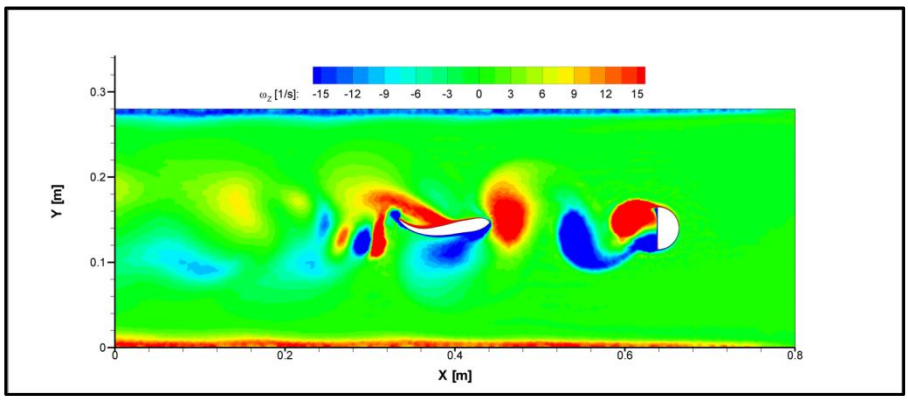
Figure 19: Flow field evolution of free-stream kinematics in the presence of a Kármán vortex street.



(D)

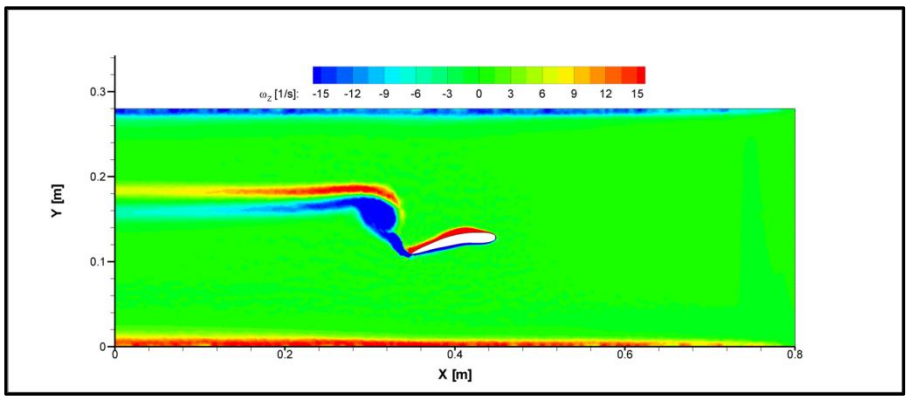


(E)

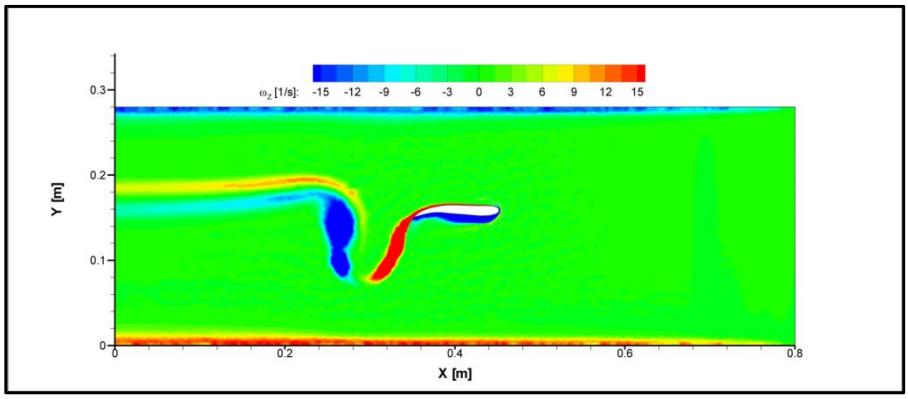


(F)

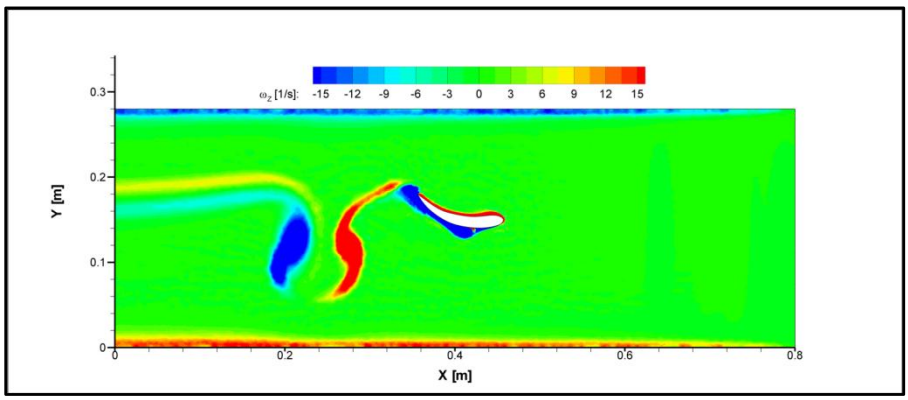
Figure 19: Continued.



(A)

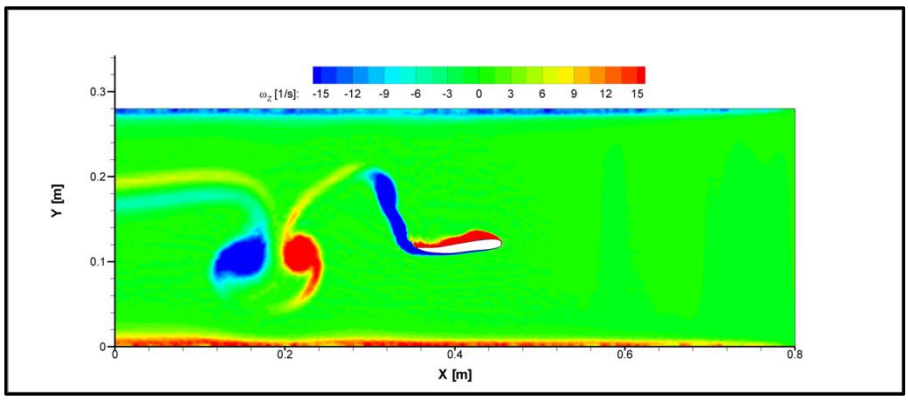


(B)

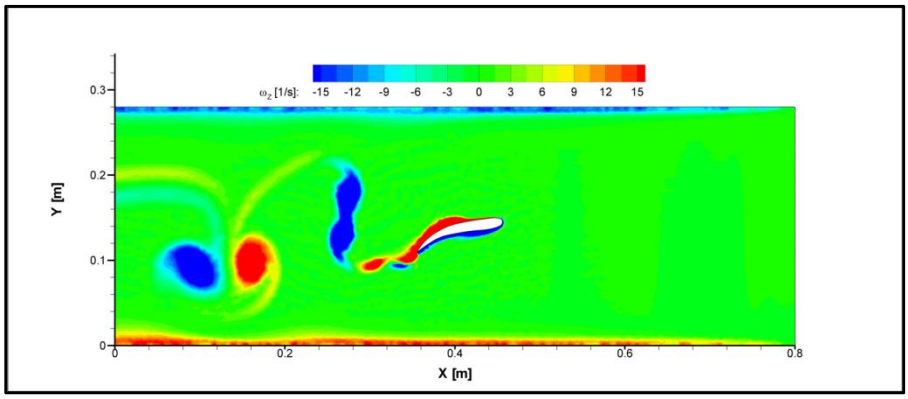


(C)

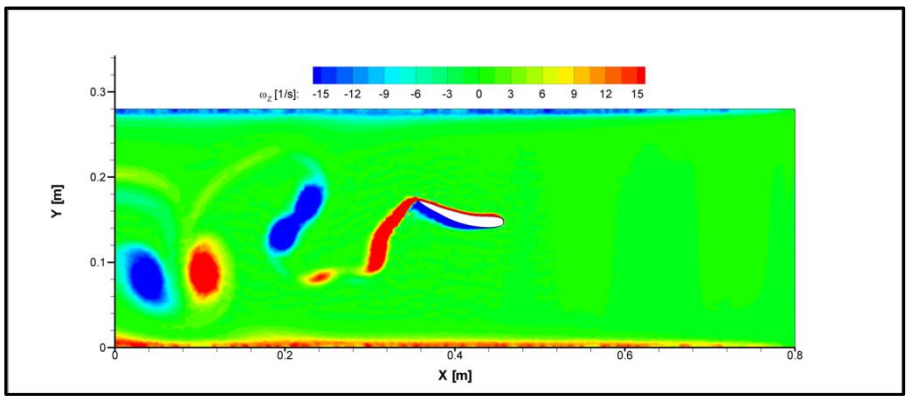
Figure 20: Flow field evolution of Kármán gait kinematics paired with an empty flume.



(D)

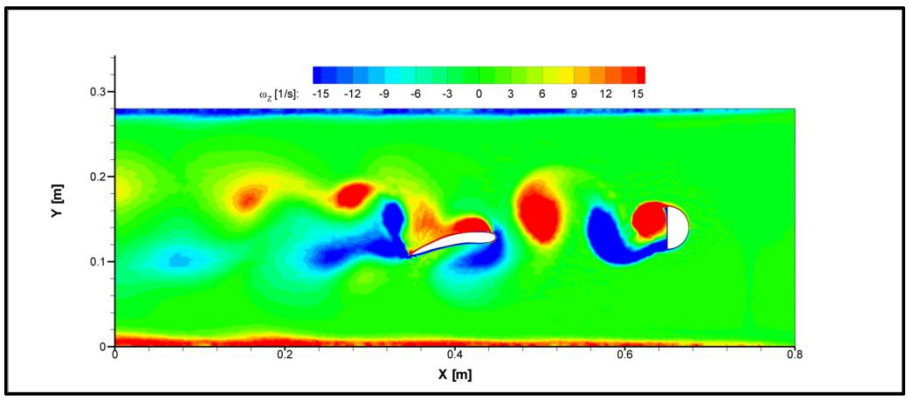


(E)

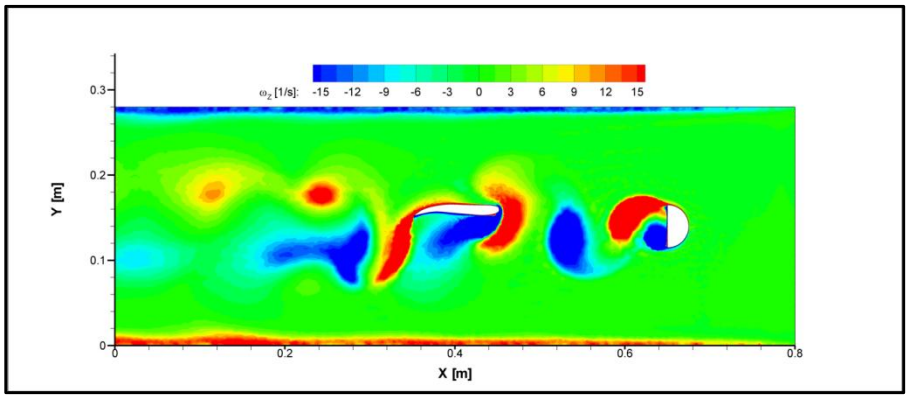


(F)

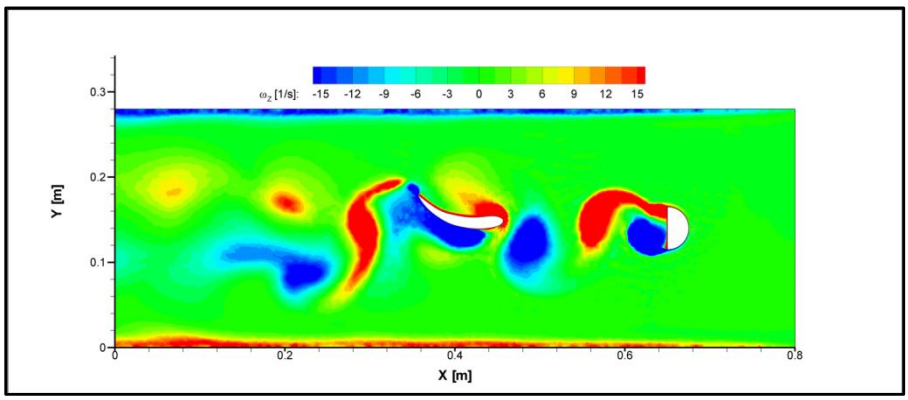
Figure 20: Continued.



(A)

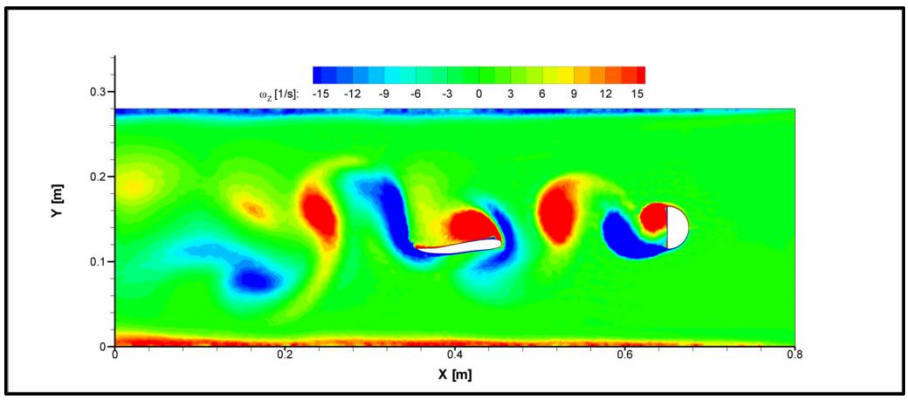


(B)

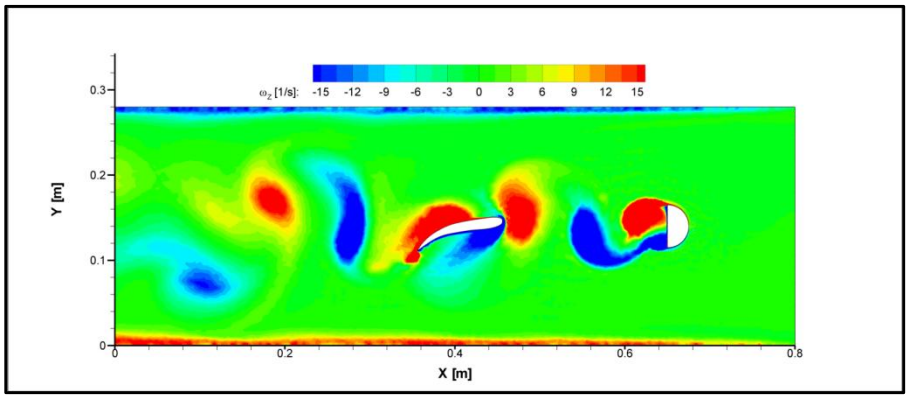


(C)

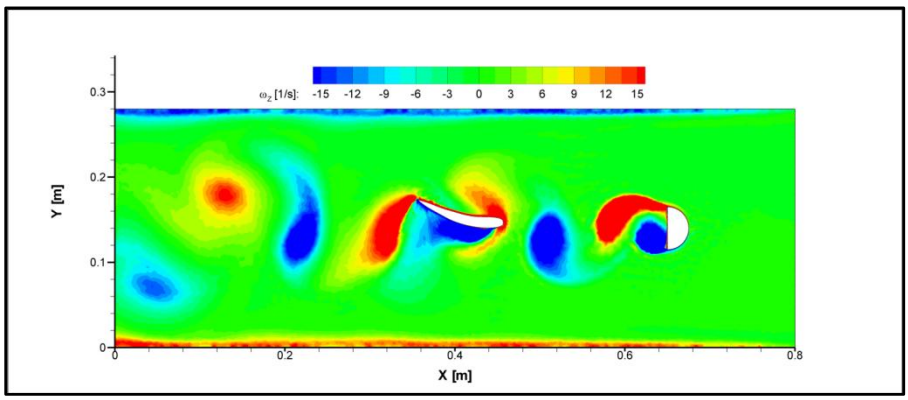
Figure 21: Flow field evolution of Kármán gait kinematics in the presence of a Kármán vortex street with incorrect initiation of motion.



(D)

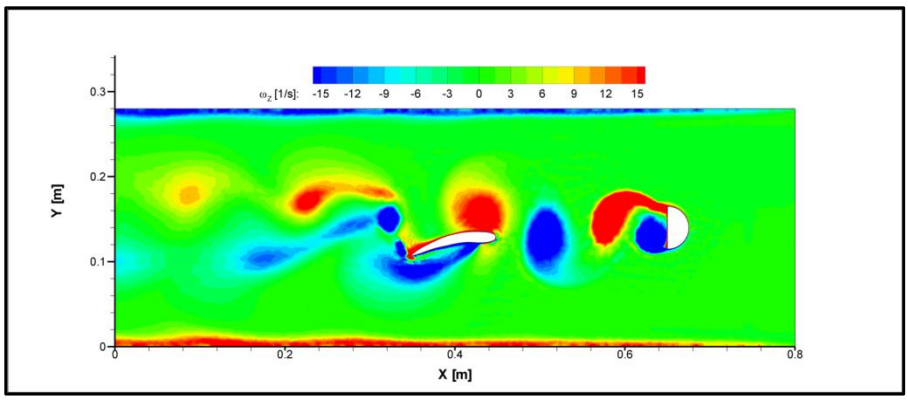


(E)

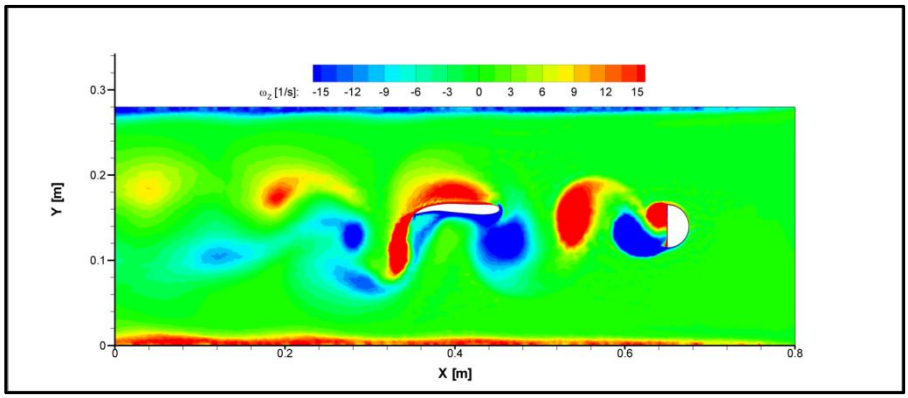


(F)

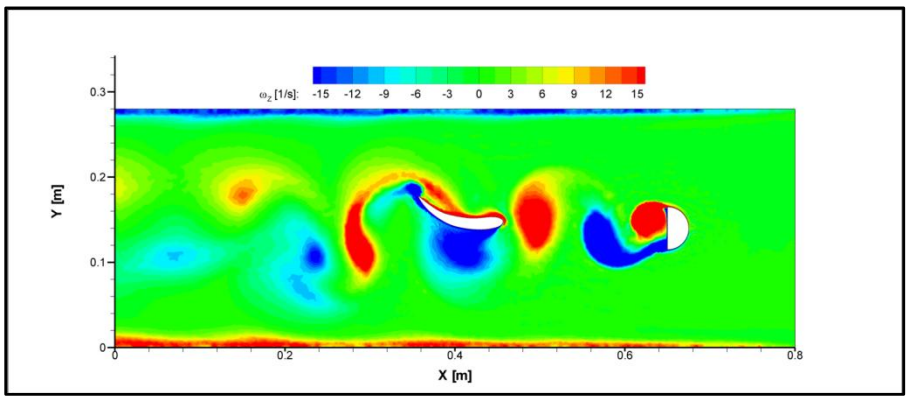
Figure 21: Continued.



(A)

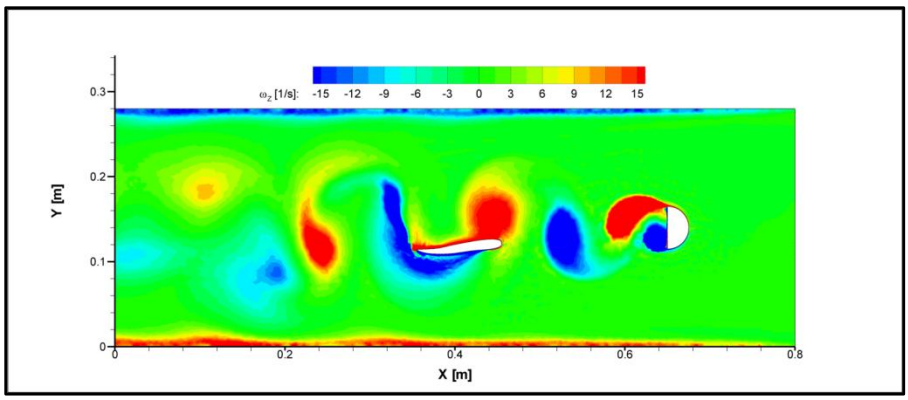


(B)

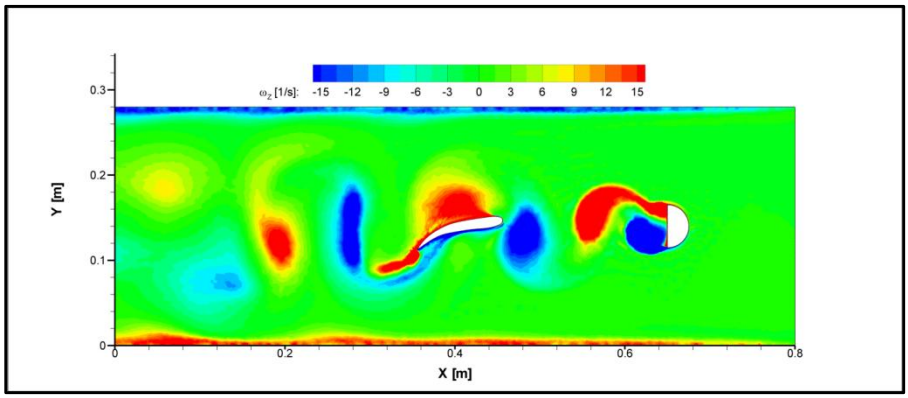


(C)

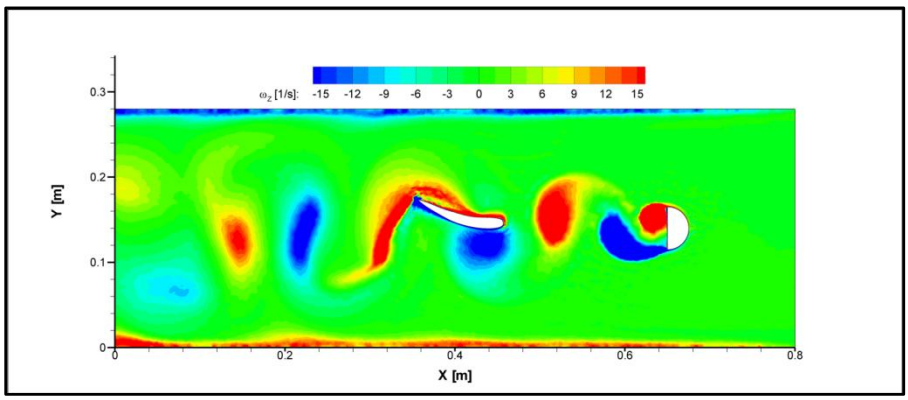
Figure 22: Flow field evolution of Kármán gait kinematics in the presence of a Kármán vortex street with correct initiation of motion.



(D)



(E)



(F)

Figure 22: Continued.

Table 3: Time-averaged data and swimming efficiency for all simulations. The experimentally observed configurations are highlighted.

Kinematics	Flume Configuration	Time Offset [s]	Time-Averaged Data for All Cycles		
			Power [W/m]	Thrust [N/m]	Efficiency
Free Stream	No Cylinder	5.000	1.19	2.61	49.8%
Karman Gait	No Cylinder	5.000	2.36	3.86	42.4%
Free Stream	Cylinder	4.0781	1.38	1.82	37.3%
Free Stream	Cylinder	4.1779	1.26	2.16	43.5%
Free Stream	Cylinder	4.2898	1.40	1.88	37.6%
Free Stream	Cylinder	4.3957	1.48	1.74	34.5%
Karman Gait	Cylinder	4.5471	4.58	3.04	23.0%
Karman Gait	Cylinder	4.4634	3.27	3.08	29.8%
Karman Gait	Cylinder	4.7340	1.31	2.91	50.1%
Karman Gait	Cylinder	4.8434	2.63	2.63	31.0%

CHAPTER V

CONCLUSION AND FUTURE WORK

Summary of Findings

In this study, eleven computational fluid dynamics simulations have been completed. The purpose of simulations was to test the performance of a new method of image analysis based CFD. Image analysis was performed on video files of fish swimming in experimental test flumes. Two videos were analyzed, each featuring a different form of fish locomotion. The two forms were free-stream locomotion and Kármán gait locomotion.

Free-stream swimming is a mode in which swim naturally in free flowing water. Fish swim using the Kármán gait in wakes containing Kármán vortex streets, where they can harness the energy contained in the vortices to increase their swimming efficiency. Experimentally, a D-section cylinder was used to create the Kármán vortex street from which the fish extracted energy.

The image analysis produced two sets of data files. Each set contained the deforming boundary of a fish swimming in a particular gait. The initial data files were used to generate four numerical grids:

1. Free-stream fish with no D-section cylinder
2. Free-stream fish with a D-section cylinder
3. Kármán gait fish with no D-section cylinder
4. Kármán gait fish with a D-section cylinder

These grids were generated to compare how the different forms of swimming modes would fair both in flow fields that they occur naturally, and in flow fields in which they do not occur naturally. It was hypothesized that the physically observed configurations (1 and 4 above) would be much more efficient that the non-physically observed

configurations (2 and 3 above). Experimental research has also shown that less energy is expended by fish in configuration 4.

Examining the flow field around each model configuration showed that the simulations worked as expected. All of the fish generated inverse Kármán gaits, as predicted. The hypothesis was backed up by the simulation results, with configuration 4 having the highest swimming efficiency, followed closely by simulation 1. Examining the flow field results from simulation 4 also reinforced the theory that Kármán gait kinematics were used to extract energy from the vortices generated by the D-section cylinder.

Future Work

Reduction of Error and Uncertainty

There are many ways in which this study could be improved. The most important, and most likely the greatest source of error in the current project, is moving the CFD simulations into the third dimension. In this project, only a cross-section of a fish was simulated, which greatly limits the usefulness of the results. However, moving the current research into the third dimension is (theoretically) fairly easy. Data files that describe all three dimensions of the fish's motion have been extracted from videos files. Unfortunately, simulating high-resolution fish swimming three dimensionally can be very computationally expensive.

The results could also be improved by moving to a more robust turbulence model. The realizable k-ε model was used in this study because it produced the most accurate two dimensional results. This would most likely not be the case if the simulations were conducted in three dimensions. Using DES or LES would presumably increase the accuracy of three dimensional simulations, compared to URANS models. However, this will also increase the run-time of each simulation significantly.

Finally, the videos that were used to generate the fish motion were of fairly low quality, which did impact the resolution of the model somewhat. Using high-resolution video files as inputs to the CFD simulations could increase the accuracy of the model.

Applications

The research described here can be expanded in many ways. This paper has shown that using image analysis based CFD is feasible and only limited by the available input videos and available computational time. Any experimental flow field that is difficult to describe using analytic functions would be an excellent candidate for this type of modeling. Research could expand into areas such as fish behavior, and fish schooling. For fish behavior, this research could be used to determine if there are characteristic 'drivers' that occur within flow fields that influence fish behavior. In regard to fish schooling, one could use this method to exactly replicate the flow field around schooling fish, something that has never been accomplished completely. Ideally, one would want to use very high-speed, high-resolution videos to capture the input data, but it has been shown that this is not completely necessary.

APPENDIX A

SOURCE CODE FOR THE DYNAMIC MESH UDF

```
/*****************************************************
node motion based on the interpolation of data points
*****************************************************/

#include "udf.h"
#include <stdio.h>
#include <math.h>

real catmull(real x0, real framenumber, real g)
{
        real a = 0.5;
        real x, wcub;

        x = x0 - framenumber;



        if (fabs(x) >= 0 && fabs(x) < 1)
        {
                wcub = (-a+2)*pow(fabs(x),3) + (a-3)*pow(fabs(x),2) + 1;
        }
        else if ( fabs(x) >= 1 && fabs(x) < 2)
        {
                wcub = -a*pow(fabs(x),3) + 5*a*pow(fabs(x),2)
                - 8*a*fabs(x) + 4*a;
        }
        else
        {
                wcub = 0;
        }

        return wcub*g;

}


DEFINE_GRID_MOTION(deform,domain,dt,time,dtime)
{
        Thread *tf = DT_THREAD(dt);
        face_t f;
        Node *v;
        real x_grid, y_grid, time_low, time_mid,
        time_high, time_lowest, time_highest;

        real deltatime, scale, timeactual;

        real timeoffset;
        int ntime_lowest=0, ntime_low=0, ntime_mid=0, ntime_high=0, ntime_highest = 0;
        int closest;
```

```
int numtimeslice, numseg, i, j;
int refinefactor, istar;

real tslice[1000] = {0}, x_low[15000]={0}, y_low[15000]={0}, u_low[15000]={0},
v_low[15000]={0};

real x_lowest[15000] = {0}, y_lowest[15000]={0}, u_lowest[15000]={0},
v_lowest[15000]={0};

real x_highest[15000] = {0}, y_highest[15000]={0}, u_highest[15000]={0},
v_highest[15000]={0};

real x_mid[15000]={0}, y_mid[15000]={0}, u_mid[15000]={0}, v_mid[15000]={0};

real x_high[15000]={0}, y_high[15000]={0}, u_high[15000]={0}, v_high[15000]={0};

real x_avet[15000]={0}, y_avet[15000]={0}, u_avet[15000]={0}, v_avet[15000]={0};

real x_avet_ext[15000]={0}, y_avet_ext[15000]={0};
real x_avet_ext2[15000]={0}, y_avet_ext2[15000]={0};

real x0, ghat_lowest_x, ghat_low_x, ghat_mid_x, ghat_high_x, ghat_highest_x;

real ghat_lowest_y, ghat_low_y, ghat_mid_y, ghat_high_y, ghat_highest_y;

real x_tmp[15000]={0}, y_tmp[15000]={0}, u_tmp[15000]={0}, v_tmp[15000]={0};

real x_diff, y_diff;

real Rold, Rclose;
FILE *timefile, *file0, *file1, *file2, *file3, *file4;

char ntime_lowf[10], ntime_midf[10], ntime_highf[10], ntime_lowestf[10],
ntime_highestf[10];

int n;

/* set deforming flag on adjacent cell zone */

SET_DEFORMING_THREAD_FLAG(THREAD_T0(tf));

time_low = 0;
time_mid = 100000000;
timeactual = time - dtime;
timeoffset = 1125;

scale = 0.028916482 / 100;
refinefactor = 1;


/*** OPEN THE TIME FILE ***/

/* open the time file */
if ((timefile = fopen("timefile.dat", "r")) == NULL)
```

```
{
        Message("Error reading file...");
}
else
{
        /* read in the time slice information */
        fscanf(timefile, "%d", &numtimeslice);
        Message("Num time slices = %d\n", numtimeslice);
        for (i = 0; i < numtimeslice; i++)
        {
                fscanf(timefile, "%lf", &tslice[i]);

                /* find time slices associated with the current time */
                if (timeactual >= tslice[i])
                {
                        time_low = tslice[i];
                }
                else if (timeactual < tslice[i] && tslice[i] < time_mid)
                {
                        time_mid = tslice[i];
                }
        }
        fclose (timefile);
}

/*** DONE OPENING THE TIMEFILE ***/


/*** FIND THE NAMES OF THE FISH BOUNDARY FILES ***/

/* initialize the time variables */
deltatime = tslice[1] - tslice[0];

time_high = time_mid + deltatime;
time_lowest = time_low - deltatime;
time_highest = time_high + deltatime;

if (time_low == 0)
{
        ntime_low = (int) ((time_low / deltatime) + 0.1);
}
else
{
        ntime_low = (int) ((time_low / deltatime) + 0.1 - timeoffset);
        ntime_lowest = ntime_low - 1;
}


ntime_mid = ntime_low + 1;
ntime_high = ntime_low + 2;
ntime_lowest = ntime_low - 1;
ntime_highest = ntime_low + 3;

sprintf(ntime_lowf, "%d", ntime_low);
```

```
sprintf(ntime_midf, "%d", ntime_mid);
sprintf(ntime_highf, "%d", ntime_high);
sprintf(ntime_lowestf, "%d", ntime_lowest);
sprintf(ntime_highestf, "%d", ntime_highest);


/*** FISH BOUNDARY FILE NAMES FOUND ***/

/*** OPEN THE FISH BOUNDARY FILES ***/

/*LOW TIME*/
if ((file1 = fopen(ntime_lowf, "r")) == NULL)
{
        Message("Error reading file1...");
}
else
{
        /* read in file1 information*/
        fscanf(file1, "%d", &numseg);
        Message("numseg = %d\n ", numseg);

        for (i = 0; i < numseg; i++)
        {
                fscanf(file1,                                   "%lf%lf%lf%lf",
                &x_low[i],&y_low[i],&u_low[i],&v_low[i]);
        }

        fclose(file1);
}


/*LOWEST TIME*/
if (ntime_low == 0)
{
        for (i = 0; i < numseg; i++)
        {
                x_lowest[i] = x_low[i];
                y_lowest[i] = y_low[i];
                u_lowest[i] = 0;
                v_lowest[i] = 0;
        }
}
else
{
        if ((file0 = fopen(ntime_lowestf, "r")) == NULL)
        {
                Message("Error reading file0...");
        }
        else
        {
                /* read in file1 information*/
                fscanf(file0, "%d", &numseg);
                Message("numseg = %d\n ", numseg);
                for (i = 0; i < numseg; i++)
```

```
                {
                        fscanf(file0, "%lf%lf%lf%lf", &x_lowest[i],&y_lowest[i],
                        &u_lowest[i],&v_lowest[i]);
                }
                fclose(file0);
        }
}


/* MID TIME */
if ((file2 = fopen(ntime_midf, "r")) == NULL)
{
        Message("Error reading file2...");
}
else
{
        /* read in file2 information */
        fscanf(file2, "%d", &numseg);
        Message("numseg = %d\n ", numseg);

        for (i = 0; i < numseg; i++)
        {
                fscanf(file2,                                   "%lf%lf%lf%lf",
                &x_mid[i],&y_mid[i],&u_mid[i],&v_mid[i]);
        }

        fclose(file2);
}

/* HIGH TIME */
if (ntime_mid == numtimeslice - 1)
{
        for (i = 0; i < numseg; i++)
        {
                x_high[i] = x_mid[i];
                y_high[i] = y_mid[i];
                u_high[i] = 0;
                v_high[i] = 0;
        }
}
else
{
        if ((file3 = fopen(ntime_highf, "r")) == NULL)
        {
                Message("Error reading file3...");
        }
        else
        {
                /* read in file3 information */
                fscanf(file3, "%d", &numseg);
                Message("numseg = %d\n ", numseg);

                for (i = 0; i < numseg; i++)
                {
```

```
                    fscanf(file3,                    "%lf%lf%lf%lf",
                    &x_high[i],&y_high[i],&u_high[i],&v_high[i]);
                }
                fclose(file3);
        }
}


/* HIGHEST TIME */
if (ntime_mid == numtimeslice - 1)
{
        for (i = 0; i < numseg; i++)
        {
                x_highest[i] = x_mid[i];
                y_highest[i] = y_mid[i];
                u_highest[i] = 0;
                v_highest[i] = 0;
        }
}
else if (ntime_high == numtimeslice - 1)
{
        for (i = 0; i < numseg; i++)
        {
                x_highest[i] = x_high[i];
                y_highest[i] = y_high[i];
                u_highest[i] = 0;
                v_highest[i] = 0;
        }
}
else
{
        if ((file4 = fopen(ntime_highestf, "r")) == NULL)
        {
                Message("Error reading file3...");
        }
        else
        {
                /* read in file3 information */
                fscanf(file4, "%d", &numseg);
                Message("numseg = %d\n ", numseg);



                for (i = 0; i < numseg; i++)
                {
                    fscanf(file4,                    "%lf%lf%lf%lf",
                    &x_highest[i],&y_highest[i],&u_highest[i],&v_highest[i]);
                }
                fclose(file4);
        }
}
/*** ALL BOUNDARY FILES OPENED ***/
```

```
/***
INTERPOLATE TO GET VALUES AT THE CURRENT TIME AND EXTENDED
TIME
***/

for (i=0; i < numseg; i++)
{

        x0 = ntime_low + (timeactual - time_low)/(time_mid-time_low);
        ghat_lowest_x = catmull(x0, ntime_lowest, x_lowest[i]);
        ghat_low_x = catmull(x0, ntime_low, x_low[i]);
        ghat_mid_x = catmull(x0, ntime_mid, x_mid[i]);
        ghat_high_x = catmull(x0, ntime_high, x_high[i]);
        ghat_lowest_y = catmull(x0, ntime_lowest, y_lowest[i]);
        ghat_low_y = catmull(x0, ntime_low, y_low[i]);
        ghat_mid_y = catmull(x0, ntime_mid, y_mid[i]);
        ghat_high_y = catmull(x0, ntime_high, y_high[i]);


        x_avet[i] = scale*(ghat_lowest_x + ghat_low_x + ghat_mid_x + ghat_high_x);

        y_avet[i] = scale*(ghat_lowest_y + ghat_low_y + ghat_mid_y + ghat_high_y);

        if (i == 0)
        {
                Message("wcub_lowest = %lf\n", ghat_lowest_x/x_lowest[i]);
                Message("wcub_low = %lf\n", ghat_low_x/x_low[i]);
                Message("wcub_mid = %lf\n", ghat_mid_x/x_mid[i]);
                Message("wcub_high = %lf\n", ghat_high_x/x_high[i]);
        }

        if ((time-time_low)/(time_mid-time_low) <= 1)
        {
                x0 = ntime_low + (time - time_low)/(time_mid-time_low);
                ghat_lowest_x = catmull(x0, ntime_lowest, x_lowest[i]);
                ghat_low_x = catmull(x0, ntime_low, x_low[i]);
                ghat_mid_x = catmull(x0, ntime_mid, x_mid[i]);
                ghat_high_x = catmull(x0, ntime_high, x_high[i]);
                ghat_lowest_y = catmull(x0, ntime_lowest, y_lowest[i]);
                ghat_low_y = catmull(x0, ntime_low, y_low[i]);
                ghat_mid_y = catmull(x0, ntime_mid, y_mid[i]);
                ghat_high_y = catmull(x0, ntime_high, y_high[i]);

                x_avet_ext[i] = scale*(ghat_lowest_x + ghat_low_x + ghat_mid_x +
                ghat_high_x);

                y_avet_ext[i] = scale*(ghat_lowest_y + ghat_low_y + ghat_mid_y +
                ghat_high_y);
        }
        else
```

```
        {
                x0 = ntime_mid + (time - time_mid)/(time_high-time_mid);
                ghat_low_x = catmull(x0, ntime_low, x_low[i]);
                ghat_mid_x = catmull(x0, ntime_mid, x_mid[i]);
                ghat_high_x = catmull(x0, ntime_high, x_high[i]);
                ghat_highest_x = catmull(x0, ntime_highest, x_highest[i]);
                ghat_low_y = catmull(x0, ntime_low, y_low[i]);
                ghat_mid_y = catmull(x0, ntime_mid, y_mid[i]);
                ghat_high_y = catmull(x0, ntime_high, y_high[i]);
                ghat_highest_y = catmull(x0, ntime_highest, y_highest[i]);
                x_avet_ext[i] = scale*(ghat_highest_x + ghat_low_x + ghat_mid_x +
                ghat_high_x);

                y_avet_ext[i] = scale*(ghat_highest_y + ghat_low_y + ghat_mid_y +
                ghat_high_y);
        }

        if (i == 0)
        {
                Message("wcub_lowest_ext = %lf\n", ghat_lowest_x/x_lowest[i]);
                Message("wcub_low_ext = %lf\n", ghat_low_x/x_low[i]);
                Message("wcub_mid_ext = %lf\n", ghat_mid_x/x_mid[i]);
                Message("wcub_high_ext = %lf\n", ghat_high_x/x_high[i]);
        }

        u_avet[i] = (x_avet_ext[i] - x_avet[i])/dtime;
        v_avet[i] = (y_avet_ext[i] - y_avet[i])/dtime;
        u_avet[i] = u_avet[i];
        v_avet[i] = v_avet[i];

}

/*** TIME INTERPOLATION COMPLETED ***/


/*** MOVE THE NODES TO THE INTERPOLATED VALUES ***/

Message("dtime = %lf, time = %lf ", dtime, timeactual);



begin_f_loop(f,tf)
{
        f_node_loop(f,tf,n)
        {
                v = F_NODE(f,tf,n);

                if (NODE_POS_NEED_UPDATE (v))
                {
                        /* indicate that node position has been updated */
                        /* so that it's not updated more than once */
                        NODE_POS_UPDATED(v);

                        x_grid = NODE_X(v);
```

```
                    y_grid = NODE_Y(v);

                    /*
                    find two data points you are located at
                    interpolate to get u, v values at current data points
                    */
                    Rold = 1000000000;
                    for (i=0; i < (numseg - 1); i++)
                    {
                            /* find the closest point */
                            x_diff = x_grid - (x_avet[i]);
                            y_diff = y_grid - (y_avet[i]);

                            Rclose = sqrt((x_diff * x_diff) + (y_diff * y_diff));

                            if (Rclose <= Rold)
                            {
                                    closest = i;
                                    Rold = Rclose;
                            }
                    }

                    NODE_X(v) = NODE_X(v) + dtime*u_avet[closest];
                    NODE_Y(v) = NODE_Y(v) + dtime*v_avet[closest];

                }
            }
    }

    end_f_loop(f,tf);
    }
```

APPENDIX B

SOURCE CODE FOR THE DATA REPORTING UDF

```
/*************************************************************************

        UDF TO CALCULATE THE EFFICIENCY OF A SWIMMING FISH

        Author: Justin W Hannon
        Date:    January - February 2011

        This UDF calculates the swimming efficiency of a fishusing thrust production, swimming
        speed, and powerinput.  This method was used by Borazjani and Sotiropoulos
        in 2008.

        This UDF works for 2D, 3D, serial, and parallel
        simulations.  The first half of this code contains
        the parallel portion of the code, while the bottom
        half contains the serial portion of the code.

        This program should work for any arbitrary boundary
        that is undergoing motion as long as the axis of
        motion is set up properly (for this simulation,
        it is the y-axis, '1').

*************************************************************************/

/* required for FLUENT macros */
#include "udf.h"

/* required for the 'BOUNDARY_FACE_GEOMETRY' function */
#include "sg.h"

/* standard math and I/O libraries */
#include <math.h>
#include <stdio.h>

/* estimated number of faces on the fish boundary - only used for serial UDF */
#define NUM_FACES 2000

/* 'ZONE_ID' is the ID number associated with the fish zone - can be found in the FLUENT BC
dialog box */
#define ZONE_ID 7

/* 'FLUID_DOMAIN' is the fluid domain, and always has a value of '1' for single phase flow */
#define FLUID_DOMAIN 1

/* this variable makes the UDF start with the initiation of motion */
#define TIME_OFFSET 2.001

/* this macro executes once per time-step, at the end of the time-step */
DEFINE_EXECUTE_AT_END(efficiency)
{
```

```
/********** BEGIN PARALLEL CODE **********/
#if PARALLEL

#if RP_NODE

/*
domain 'd' is the fluid domain (ID = 1) and its address is looked up by the function
'Get_Domain'
*/
Domain *d;

/* face threads and indentifiers */
Thread *f_thread;
face_t f;

/*
the below variables are intialized by the function 'BOUNDARY_FACE_GEOMETRY' -
only the variable 'A' is used in this UDF
*/
real A[ND_ND], ds, es[ND_ND], A_by_es, dr0[ND_ND];

/* temporary variable to store the y-location of the face centroids */
real x[ND_ND];

#endif

/*
arrays used to pass data as follows: node_1, node_2, ...
, node_n-1 -> node_0 -> host
*/
real *y_array = NULL;
real *p_array = NULL;
real *tx_array = NULL;
real *ty_array = NULL;
real *Ax_array = NULL;
real *Ay_array = NULL;

/*
        stores destination of the data passing arrays,
        and loop through nodes when host is receiving data
*/
int pe;

/*
        variable that tells destination processes the
        length of the arrays they will receive
*/
int size;

#if RP_HOST

/* counter variable */
int i;
```

```
/* initialize variables for the results file */
real force = 0;
real thrust = 0;
real drag = 0;
real power = 0;

/* define current flow time */
real time = CURRENT_TIME;

/* stores the data from the current time-step */
real *y_new = NULL;
real *p_new = NULL;
real *tx_new = NULL;
real *ty_new = NULL;
real *Ax_new = NULL;
real *Ay_new = NULL;

/* stores the data from the previous time-step */
real *y_old = NULL;
real *p_old = NULL;
real *tx_old = NULL;
real *ty_old = NULL;
real *Ax_old = NULL;
real *Ay_old = NULL;

/* counts the number of fish boundary faces   */
/* when host is receiving data from each node */
int total_num_faces = 0;

/* file pointers for all of the data files */
FILE *previous_ts_data;
FILE *current_ts_data;
FILE *results;

#endif

#if RP_NODE

/* retrieve fluid domain address and look up the face thread */
d = Get_Domain(FLUID_DOMAIN);
f_thread = Lookup_Thread(d,ZONE_ID);

/* determine the number of elements in the current partition */
size = THREAD_N_ELEMENTS_INT(f_thread);

/* allocate memory for the data arrays */
y_array = (real *)malloc(size * sizeof(real));
p_array = (real *)malloc(size * sizeof(real));
tx_array = (real *)malloc(size * sizeof(real));
ty_array = (real *)malloc(size * sizeof(real));
Ax_array = (real *)malloc(size * sizeof(real));
Ay_array = (real *)malloc(size * sizeof(real));

/* loop through each face on the fish boundary */
```

```
begin_f_loop(f,f_thread)
{
        /*
        makes sure that partition boundary faces are only counted once
        */
        if (PRINCIPAL_FACE_P(f,f_thread))
        {
                /*
                return the face area normals as array 'A' (in addition to other unused
                variables)
                */
                BOUNDARY_FACE_GEOMETRY(f,f_thread,A,ds,es,A_by_es,dr0);

                /* return the face centroid as array 'x' */
                F_CENTROID(x,f,f_thread);

                /*
                store all of the relevant data in the following arrays
                */

                /*
                the variables are: centroid y-location, face pressure,
                x-direction shear force, y-direction shear force, x-normal area, and y-
                normal area
                */
                y_array[f] = x[1];

                p_array[f] = F_P(f,f_thread);

                tx_array[f] = F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[0];
                ty_array[f] = F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[1];

                Ax_array[f] = A[0];
                Ay_array[f] = A[1];
        }
}
end_f_loop(f,f_thread)

/* 'pe' stores the destination process (node_0 or host) */
pe = (I_AM_NODE_ZERO_P) ? node_host : node_zero;

/* tell the destination process the length of arrays to expect */
PRF_CSEND_INT(pe, &size, 1, myid);

/* send the data arrays to the destination process (node_0 or host) */
PRF_CSEND_REAL(pe, y_array, size, myid);
PRF_CSEND_REAL(pe, p_array, size, myid);
PRF_CSEND_REAL(pe, tx_array, size, myid);
PRF_CSEND_REAL(pe, ty_array, size, myid);
PRF_CSEND_REAL(pe, Ax_array, size, myid);
PRF_CSEND_REAL(pe, Ay_array, size, myid);

/* release the data arrays after sending them */
free(y_array);
```

```
free(p_array);
free(tx_array);
free(ty_array);
free(Ax_array);
free(Ay_array);

/*
node_0 now collects the data from the other nodes and sends it to host */
if (I_AM_NODE_ZERO_P)
{
        /*
        loop through receiving data from node_1, node_2, ..., node_n-1
        */
        compute_node_loop_not_zero (pe)
        {
                /* receive the length of the arrays */
                PRF_CRECV_INT(pe, &size, 1, pe);

                /* allocate memory for the data arrays */
                y_array = (real *)malloc(size * sizeof(real));
                p_array = (real *)malloc(size * sizeof(real));
                tx_array = (real *)malloc(size * sizeof(real));
                ty_array = (real *)malloc(size * sizeof(real));
                Ax_array = (real *)malloc(size * sizeof(real));
                Ay_array = (real *)malloc(size * sizeof(real));

                /*
                receive the data arrays from node_1, node_2, ..., node_n-1 */
                PRF_CRECV_REAL(pe, y_array, size, pe);
                PRF_CRECV_REAL(pe, p_array, size, pe);
                PRF_CRECV_REAL(pe, tx_array, size, pe);
                PRF_CRECV_REAL(pe, ty_array, size, pe);
                PRF_CRECV_REAL(pe, Ax_array, size, pe);
                PRF_CRECV_REAL(pe, Ay_array, size, pe);

                /* tell the host the length of arrays to expect */
                PRF_CSEND_INT(node_host, &size, 1, myid);

                /* send the data arrays to host */
                PRF_CSEND_REAL(node_host, y_array, size, myid);
                PRF_CSEND_REAL(node_host, p_array, size, myid);
                PRF_CSEND_REAL(node_host, tx_array, size, myid);
                PRF_CSEND_REAL(node_host, ty_array, size, myid);
                PRF_CSEND_REAL(node_host, Ax_array, size, myid);
                PRF_CSEND_REAL(node_host, Ay_array, size, myid);

                /* release the data arrays after sending them */
                free(y_array);
                free(p_array);
                free(tx_array);
                free(ty_array);
                free(Ax_array);
                free(Ay_array);
        }
```

```
}

#endif /* RP_NODE */

#if RP_HOST

if ((current_ts_data = fopen("data_t-0.dat", "w")) == NULL)
{
        Message("Error opening 'data_t-0.dat' for writing...");
}

else
{
        /*
        loop through receiving data from node_0, node_1, node_2, ...,
        node_n-1
        */
        compute_node_loop(pe)
        {
                /* receive the length of the arrays from node_0 */
                PRF_CRECV_INT(node_zero, &size, 1, node_zero);

                /* allocate memory for the data arrays */
                y_array = (real *)malloc(size * sizeof(real));
                p_array = (real *)malloc(size * sizeof(real));
                tx_array = (real *)malloc(size * sizeof(real));
                ty_array = (real *)malloc(size * sizeof(real));
                Ax_array = (real *)malloc(size * sizeof(real));
                Ay_array = (real *)malloc(size * sizeof(real));

                /* receive the data arrays from node_0 */
                PRF_CRECV_REAL(node_zero, y_array, size, node_zero);
                PRF_CRECV_REAL(node_zero, p_array, size, node_zero);
                PRF_CRECV_REAL(node_zero, tx_array, size, node_zero);
                PRF_CRECV_REAL(node_zero, ty_array, size, node_zero);
                PRF_CRECV_REAL(node_zero, Ax_array, size, node_zero);
                PRF_CRECV_REAL(node_zero, Ay_array, size, node_zero);

                /*
                loop through the arrays and write out the data
                to 'data_t-0.dat'
                */
                for (i = 0; i < size; i++)
                {
                        fprintf(current_ts_data,        "%le\t%le\t%le\t%le\t%le\t%le\n",
                        y_array[i], p_array[i], tx_array[i], ty_array[i], Ax_array[i],
                        Ay_array[i]);
                }

                /* release the data arrays after wrting them */
                free(y_array);
                free(p_array);
                free(tx_array);
                free(ty_array);
```

```
                free(Ax_array);
                free(Ay_array);

                /*
                determine the total number of faces on the fish boundary */
                total_num_faces += size;
        }
        fclose(current_ts_data);
}

/*
in this section of code the UDF will read
in the data that was just written out
*/

/*
        this will be done for two reasons:

        1) if motion hasn't begun yet, the UDF will write out the current data as
        theprevious time-step data and exit - no comparison is made through time

        2) if motion has begun, the UDF will read in the current data to ensure that the
        data from the previous time-step and current data are aligned in their respective
        arrays
*/

if ((current_ts_data = fopen("data_t-0.dat", "r")) == NULL)
{
        Message("Error opening 'data_t-0.dat' for reading...");
}

else
{
        /* allocate memory for the previous time-step data */
        y_new = (real *)malloc(total_num_faces * sizeof(real));
        p_new = (real *)malloc(total_num_faces * sizeof(real));
        tx_new = (real *)malloc(total_num_faces * sizeof(real));
        ty_new = (real *)malloc(total_num_faces * sizeof(real));
        Ax_new = (real *)malloc(total_num_faces * sizeof(real));
        Ay_new = (real *)malloc(total_num_faces * sizeof(real));

        /* loop through 'data_t-0.dat' to read in the current time-step data */
        for (i = 0; i < total_num_faces; i++)
        {
                fscanf(current_ts_data,        "%le%le%le%le%le%le",        &y_new[i],
                &p_new[i], &tx_new[i], &ty_new[i], &Ax_new[i], &Ay_new[i]);
        }
        fclose(current_ts_data);
}

/*      BEGIN THE PROCEDURE TO CALCULATE THE OUTPUT DATA */

/*
        this is only done after one time-step has been completed,
```

```
        otherwise there will be no data to read into the $_old arrays
*/
if (time > CURRENT_TIMESTEP + TIME_OFFSET)
{
        /* allocate memory for the previous time-step data */
        y_old = (real *)malloc(total_num_faces * sizeof(real));
        p_old = (real *)malloc(total_num_faces * sizeof(real));
        tx_old = (real *)malloc(total_num_faces * sizeof(real));
        ty_old = (real *)malloc(total_num_faces * sizeof(real));
        Ax_old = (real *)malloc(total_num_faces * sizeof(real));
        Ay_old = (real *)malloc(total_num_faces * sizeof(real));

        if ((previous_ts_data = fopen("data_t-dt.dat", "r")) == NULL)
        {
                Message("Error opening 'data_t-dt.dat' for reading...");
        }

        else
        {
                /*
                loop through 'data_t-dt.dat' to read in the
                previous time-step data
                */
                for (i = 0; i < total_num_faces; i++)
                {
                        fscanf(previous_ts_data,  "%le%le%le%le%le",  &y_old[i],
                        &p_old[i], &tx_old[i], &ty_old[i], &Ax_old[i], &Ay_old[i]);
                }
                fclose(previous_ts_data);
        }

        /* loop through the data arrays and calculate output data */
        for (i = 0; i < total_num_faces; i++)
        {
                force += -p_new[i] * Ax_new[i] + tx_new[i];

                thrust += 0.5 * (-p_new[i] * Ax_new[i] + fabs(p_new[i] * Ax_new[i])) +
                0.5 * (tx_new[i] + fabs(tx_new[i]));

                drag += -(0.5 * (-p_new[i] * Ax_new[i] - fabs(p_new[i] * Ax_new[i])) +
                0.5 * (tx_new[i] - fabs(tx_new[i])));

                power += ((y_new[i] - y_old[i]) / CURRENT_TIMESTEP) *
                (((-(p_new[i] + p_old[i])) / 2) * (((Ay_new[i] + Ay_old[i]) / 2)) +
                (((ty_new[i] + ty_old[i]) / 2)));
        }

        if ((results = fopen("results.dat", "a+")) == NULL)
        {
                Message("Error opening 'results.dat' for reading...");
        }

        else
        {
```

```
                /* write out the final data */
                fprintf(results, "%le\t%le\t%le\t%le\t%le\n", time, force, thrust, drag,
                power);
                fclose(results);
        }

        /* release memory that held the previous time-step data */
        free(y_old);
        free(p_old);
        free(tx_old);
        free(ty_old);
        free(Ax_old);
        free(Ay_old);
}

/* this section activates when motion has not begun */
else
{
        if ((results = fopen("results.dat", "w")) == NULL)
        {
                Message("Error opening 'results.dat' for writing...");
        }

        /* write out headers for 'results.dat' */
        else
        {
                fprintf(results, "%s\t\t%s\t\t%s\t\t%s\t\t%s\n", "Time", "Force", "Thrust",
                "Drag", "Power");
                fclose(results);
        }
}

if ((previous_ts_data = fopen("data_t-dt.dat", "w")) == NULL)
{
        Message("Error opening 'data_t-dt.dat' for writing...");
}

else
{
        /*
        write out the current time-step data as 'data_t-dt.dat'
        so that we have input data for the next time-step
        */
        for (i = 0; i < total_num_faces; i++)
        {
                fprintf(previous_ts_data, "%le\t%le\t%le\t%le\t%le\t%le\n", y_new[i],
                p_new[i], tx_new[i], ty_new[i], Ax_new[i], Ay_new[i]);
        }
        fclose(previous_ts_data);

        /* release memory that held the current time-step data */
        free(y_new);
        free(p_new);
        free(tx_new);
```

```
        free(ty_new);
        free(Ax_new);
        free(Ay_new);
}

#endif /* RP_HOST */

#endif
/********** END PARALLEL CODE **********/


/********** BEGIN SERIAL CODE **********/
#if !PARALLEL

/*
domain 'd' is the fluid domain (ID = 1) and its address is looked up by the function 'Get_Domain' */
Domain *d;

/* intialize face threads and indentifiers */
Thread *f_thread = 0;
face_t f;

/* initialize variables for the results file */
real pressure_force = 0;
real shear_force = 0;
real total_force = 0;
real thrust = 0;
real drag = 0;
real power = 0;
real efficiency = 0;

/*
the below variables are intialized by the function 'BOUNDARY_FACE_GEOMETRY' - only the variable 'A' is used in this UDF */
real A[ND_ND];
real ds, es[ND_ND], A_by_es, dr0[ND_ND];

/* define current flow time */
real time = CURRENT_TIME;

/*
these variables are stored in a temporary file to retain information from the previous time-step
*/

/*
they are: face centroid (x,y), y-component of the face centroid, face pressure, y-component of the face shear force, and y-component of the face area normal
*/
real x[ND_ND];
real y_old[NUM_FACES];
real p_old[NUM_FACES];
real t_old[NUM_FACES];
```

```
real A_old[NUM_FACES];

/* counter variable */
int i;

/* declare and open 'previous_ts_data' and 'results' for APPENDING */
FILE *previous_ts_data;
FILE *results;

/* file 'data_t-dt.dat' will contain data from the previous time-step */
if ((previous_ts_data = fopen("data_t-dt.dat", "a+")) == NULL)
{
        Message("Error opening data_t-dt.dat");
}

else if ((results = fopen("results.dat", "a+")) == NULL)
{
        Message("Error opening results.data...");
}

else
{
        /* retrieve fluid domain address and look up the face thread */
        d = Get_Domain(FLUID_DOMAIN);
        f_thread = Lookup_Thread(d,ZONE_ID);

        /*
        if this is the first time-step, print out headers
        for 'results.dat' and write out the initial 'data_t-dt.dat' file
        */
        if (time <= CURRENT_TIMESTEP + TIME_OFFSET)
        {
                /* write the headers to 'results.dat' and close it*/
                fprintf(results, "%s\t\t%s\t\t%s\t\t%s\t\t%s\n", "Time", "Total Force",
                "Thrust", "Drag", "Power", "Efficiency");
                fclose(results);

                /* begin loop across the fish boundary faces */
                begin_f_loop(f,f_thread)
                {
                        /*
                        returns the face area normals as array 'A' (in addition to other
                        unused variables)
                        */

                        BOUNDARY_FACE_GEOMETRY
                        (f,f_thread,A,ds,es,A_by_es,dr0);

                        /* returns the face centroid as array 'x' */
                        F_CENTROID(x,f,f_thread);

                        /* writes out data that will be used in the next time-step to
                        calculate lateral power input */
```

```
                fprintf(previous_ts_data, "%le\t%le\t%le\t%le\n", x[1],
                F_P(f,f_thread),
                F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[1],
                A[1]);
        }
        end_f_loop(f,f_thread)
}


/*
this else statement is carried out every time-step, except the first
*/
else
{
        /*
        loops through 'data_t-dt.dat' file and
        reads in variables from the previous time-step
        */
        /*
        the variable 'NUM_FACES' is an estimate and will
        need to be changed for refined grids
        */
        for (i = 0; i < NUM_FACES; i++)
        {
                fscanf(previous_ts_data, "%le%le%le%le", &y_old[i],
                &p_old[i], &t_old[i], &A_old[i]);
        }
        fclose(previous_ts_data);

        /*
        reset counter 'i' to zero so it can be used in the following face loop
        */
        i = 0;

        /*
        loop across all fish boundary faces and calculate force, thrust, drag, and
        power
        */
        begin_f_loop(f,f_thread)
        {
                /*
                returns the face area normals as array 'A' (in addition to other
                unused variables)
                */

                BOUNDARY_FACE_GEOMETRY
                (f,f_thread,A,ds,es,A_by_es,dr0);

                /* returns the face centroid as array 'x' */
                F_CENTROID(x,f,f_thread);
```

```
        /*
        following five lines calculate forces and power according to
        Borazjani and Sotiropoulos, 2008
        */
        /*
        average values from the current time-step and previous time-step
        are used in the calculation of power
        */
        pressure_force += -F_P(f,f_thread) * A[0];

        shear_force +=
        F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[0];

        thrust += 0.5 * (-F_P(f,f_thread) * A[0] + fabs(F_P(f,f_thread) *
        A[0])) + 0.5 *
        (F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[0] +
        fabs(F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[0])
        );

        drag += -(0.5 * (-F_P(f,f_thread) * A[0] - fabs(F_P(f,f_thread) *
        A[0])) + 0.5 *
        (F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[0] -
        fabs(F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[0])
        ));

        power += ((x[1] - y_old[i]) / CURRENT_TIMESTEP) * ( (-
        (F_P(f,f_thread) + p_old[i]) * (A[1] + A_old[i]) / 4) +
        (F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[1] +
        t_old[i]) / 2);

        /*
        increase the counter 'i' so that the when the next loop begins, the
        previous time-step data is being accessed at the correct locations
        in the arrays (for power calculation only)
        */
        i++;
}
end_f_loop(f,f_thread)

/*
sum forces and calculate efficiency according to Boranzjani and
Sotiropoulos, 2008
*/
total_force = pressure_force + shear_force;

/* write out the results and close the file */
fprintf(results, "%le\t%le\t%le\t%le\t%le\t\n", time, total_force, thrust,
drag, power);
fclose(results);

/*
overwrite the file 'data_t-dt.dat' with updated values for the centroid
location, pressure, shear force, and area normal
*/
```

```
                        /*
                        this data will be used in the next time-step for power calculations
                        */
                        if ((previous_ts_data = fopen("data_t-dt.dat", "w")) == NULL)
                        {
                                Message("Error opening previous time-step file for writing...");
                        }

                        else
                        {
                                begin_f_loop(f,f_thread)
                                {


                                        BOUNDARY_FACE_GEOMETRY
                                        (f,f_thread,A,ds,es,A_by_es,dr0);

                                        F_CENTROID(x,f,f_thread);

                                        fprintf(previous_ts_data,  "%le\t%le\t%le\t%le\n",  x[1],
                                        F_P(f,f_thread),
                                        F_STORAGE_R_N3V(f,f_thread,SV_WALL_SHEAR)[
                                        1], A[1]);
                                }
                                end_f_loop(f,f_thread)

                                fclose(previous_ts_data);
                        }
                }
        }

        #endif
        /********** END SERIAL CODE **********/
}
```

REFERENCES

Adkins, D., & Yan., Y. Y. (2006). CFD simulation of fish-like body moving in viscous liquid. *J. Bionic Eng., 3*, 147-153.

Anderson, J. M., Streitlien, K., Barrett, D. S., & Triantafyllou, M. S. (1997). Oscillating foils of high propulsive efficiency. *Journal of Fluid Mechanics*, 41-72.

ANSYS, Inc. (2009, April). ANSYS FLUENT 12.0 User's Guide. Canonsburg, Pennsylvania, United State of America.

Barrett, D. S., Triantafyllou, M. S., Yue, D. K., Grosenbaugh, M. A., & Wolfgang, M. J. (1999). Drag reduction in fish-like locomotion. *J. Fluid Mech., 392*, 183-212.

Beal, D. N., Hover, F. S., Triantafyllou, M. S., Liao, J. C., & Lauder, G. V. (2006). Passive propulsion in vortex wakes. *J. Fluid Mech., 549*, 385-402.

Benim, A. C., Pasqualotto, E., & Suh, S. H. (2008). Modelling turbulent flow past a circular cylinder by RANS, URANS, LES and DES. *Progress in Comp. Fluid Dyn., 8*, 299-307.

Borazjani, I., & Sotiropoulos, F. (2008). Numerical investigation of the hydrodynamics of carangiform swimming in the transitional and inertial flow regimes. *J. Exp. Biology, 211*, 1541-1558.

Borazjani, I., & Sotiropoulos, F. (2010). On the role of form and kinematics on the hydrodynamics of self-propelled body/caudal fin swimming. *J. Exp. Biology, 213*, 89-107.

Burger, W., & Burge, M. J. (2008). *Digital Image Processing: An Algorithmic Introduction Using Java.* Springer.

Cook, C. L., & Coughlin, D. J. (2010). Rainbow trout Oncorhynchus mykiss consume less energy when swimming near obstructions. *J. Fish Biology, 77*, 1716-1723.

Garvin, J. W., Kureksiz, O., Breczinski, P. J., & Garvin, M. K. (2011). A method for an image analysis-based computational fluid dynamics simulation of moving fish. *Under Review*.

Goodwin, R., Nestler, J. M., Anderson, J. J., Weber, L. J., & Loucks, D. P. (2006). Forecasting 3-D fish movement behavior using a Eulerian-Lagrangian-agent method (ELAM). *Ecological Modelling, 192*, 197-223.

Koochesfahani, M. M. (1989). Vortical patterns in the wake of an oscillating airfoil. *AIAA Journal, 27*, 1200-1205.

Lai, J. C., & Platzer, M. F. (1999). Jet characteristics of a plunging airfoil. *AIAA Journal, 37*, 1529-1537.

Liao, J. C. (2004). Neuromuscular control of trout swimming in a vortex street: implications for energy economgy during the Karman gait. *J. Exp. Biology, 207*, 3495-3506.

Liao, J. C. (2007). A review of fish swimming mechanics and behavior in altered flows. *Phil. Trans. R. Soc. B, 362*, 1973-1993.

Liao, J. C., Beal, D. N., Lauder, G. V., & Triantafyllou, M. S. (2003). The Karman gait: novel body kinematics of rainbow trout swimming in a vortex street. *J. Exp. Biology, 206*, 1059-1073.

Mittal, R. (2004). Computational modeling in biogydrodynamics: trends, challenges, and recent advances. *IEEE J. Oceanic Eng.*, 595-604.

Przybilla, A., Kunze, S., Rudert, A., Bleckmann, H., & Brucker, C. (2010). entraining in trout: a behavioural and hydrodynamic analysis. *J. Exp. Biology, 213*, 2976-2986.

Schnipper, T., Andersen, A., & Bohr, T. (2009). Vortex wakes of a flapping foil. *J. Fluid Mech., 633*, 411-423.

Shen, L., Zhang, X., Yue, D. K., & Triantafyllou, M. S. (2003). Turbulent flow over a flexible wall undergoing a streamwise travelling wave motion. *J. Fluid Mech., 484*, 197-221.

Shih, T.-H., Liou, W. W., Shabbir, A., Yanh, Z., & Zhu, J. (1995). A new k-ε eddy viscosity model for high Reynolds number turbulent flows. *Computers Fluids, 24*, 227-238.

Spalart, P. R. (2009). Detached-eddy simulation. *Annu. Rev. Fluid Mech., 41*, 181-202.

Streitlien, K., & Triantafyllou, G. S. (1998). On thrust estimates for flapping foils. *J. Fluids and Structures, 12*, 47-55.

Taguchi, M., & Liao, J. C. (2011). Rainbow trout consume less oxygen in turbulence: the energetics of swimming behaviors at different speeds. *J. Exp. Biology, 214*, 1428-1436.

Tsinober, A. (2001). *Informal Introduction to Turbulence.* Secaucus, NJ: Kluwer Academic Publishers.

White, F. M. (2008). *Fluid Mechanics* (6th ed.). New York City: McGraw-Hill.