Spring 5-12-2022

A New Model for Predicting the Drag and Lift Forces of Turbulent Newtonian Flow on Arbitrarily Shaped Shells on the Seafloor

Carley R. Walker
University of Southern Mississippi

James V. Lambers
University of Southern Mississippi

Julian Simeonov
US Naval Research Laboratory

A NEW MODEL FOR PREDICTING THE DRAG AND LIFT FORCES OF TURBULENT NEWTONIAN FLOW ON ARBITRARILY SHAPED SHELLS ON THE SEAFLOOR

by

Carley Rene Walker

A Dissertation
Submitted to the Graduate School,
the College of Arts and Sciences
and the School of Mathematics and Natural Sciences
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Approved by:

Dr. James V. Lambers, Committee Chair
Dr. Haiyan Tian
Dr. Zhifu Xie
Dr. Huiqing Zhu
Dr. Julian Simeonov

May 2022

ABSTRACT

Currently, all forecasts of currents, waves, and seafloor evolution are limited by a lack of fundamental knowledge and the parameterization of small-scale processes at the seafloor-ocean interface. Commonly used Euler-Lagrange models for sediment transport require parameterizations of the drag and lift forces acting on the particles. However, current parameterizations for these forces only work for spherical particles. In this dissertation we propose a new method for predicting the drag and lift forces on arbitrarily shaped objects at arbitrary orientations with respect to the direction of flow that will ultimately provide models for predicting the sediment sorting processes that lead to the variability of shell fragments on inner shelf seafloors. We wish to develop the drag force parameterization specifically for a limpet shell through the linear regression of force estimated from high-fidelity Reynolds-averaged Navier-Stokes (RANS) simulations in OpenFOAM.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the incredible support and invaluable assistance of my advisor and professor, Dr. James Lambers, and my mentor, Dr. Julian Simeonov. I want to thank Dr. Simeonov for his incredible patience and guidance throughout my research —I could not have succeeded without his knowledge and encouragement. I also want to thank Dr. Lambers for his overwhelming devotion to my success. My entire college career would not have been nearly as fruitful, inspiring, and gratifying without him.

I also want to thank the rest of my dissertation committee, Dr. Haiyan Tian, Dr. Huiqing Zhu, and Dr. Zhifu Xie, for their wise counsel and ardor regarding my work.

Finally, I want to thank my mom, dad, and my sisters. They've always believed in me, and even during tough times, they supported my decisions and offered nothing but love and consolation.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

Table

LIST OF ABBREVIATIONS

RANS - Reynolds-averaged Navier-Stokes
SST - Shear Stress Transport
CSF - Corey Shape Factor

Chapter 1

INTRODUCTION

1.1 The Problem

Currently, seafloor topography has been considered static for operational purposes in many different applications. Predictive models at small scales require more accurate data than is currently available, especially for predicting the bottom transport of irregularly shaped sediments, such as a shell or fragmented pieces of a shell. Predictive models at large scales, usually kilometers, are more commonly used, but due to their treatment of sediment as a continuum, they inaccurately predict seafloor evolution.

Euler-Lagrange models are used to track the dynamics of individual particles, but these models require separate parameterizations of the hydrodynamic forces acting on these particles. The ephemeral nature of the seafloor creates difficulties for performing sufficient field experiments for this particular problem, and the large variety of shell shapes and possible orientations makes defining a shape factor for the parameterization of the drag and lift force coefficients a daunting task. Therefore, we wish to create a general model for the parameterization of the hydrodynamic forces acting on a shell or shell fragment to be used in these small-scale Euler-Lagrange forecasting models for the benefit of overall seafloor evolution prediction.

1.2 The Purpose of a New Method

This dissertation is motivated by the need for: (a) an effective and robust parameterization of the drag and lift coefficients on an arbitrarily shaped shell or shell fragment on the seafloor and (b) a more efficient and accurate method for determining the drag and lift forces on a shell or shell fragment caused by turbulent flow in a Newtonian fluid.

We will perform Reynolds-Averaged Navier-Stokes (RANS) simulations on a limpet patella shell using the shear stress transport (SST) $k-\omega$ turbulence model on 103 different orientations and seven different Reynolds numbers ranging from 12,000 to 100,000. The simulated results for our drag and lift coefficients will be regressed against the components of velocity in the object frame of reference, and a model will be determined from these regressions. In this dissertation, we only show results for a Reynolds number of 12,000 and

one shape factor. From these simulations, a better understanding of the forces acting upon these oriented shells will aid in the longterm goal of accurately predicting sediment transport and seafloor roughness. With this proposed model, one will have the ability to accurately predict the drag and lift force on a limpet shell with only a few prior-known parameters, including mainstream flow velocity, fluid density, and particle surface area, and eventually, one will be able to predict these hydrodynamic forces on an arbitrarily shaped shell with these same parameters including a shape factor.

In Chapter 2 we will introduce previous numerical methods and drag force equations used for Euler-Lagrange model predictions and previous work on drag forces acting on nonspherical particles, limpet shells, and flat plates with varying angles of attack. In Chapter 3 we will give an introduction to the SST $k-\omega$ turbulence model and our new drag force theory. In Chapter 4 we will give a thorough walkthrough of setting up a comparative study using the SST $k-\omega$ turbulence model in OpenFOAM and discuss our results. In Chapter 5 we will give another walkthrough of implementing our new theory to a limpet shell and discuss our results. In Chapter 6 we will discuss establishing a shape factor tensor into our force coefficients, and finally, in Chapter 7, we will give a conclusion.

Chapter 2

BACKGROUND

2.1 Previous Numerical Methods used for Euler-Lagrange Forecasting Models

O. Ayala et al. [3] simulated turbulent collisions of hydrodynamically-interacting particles using the following predictive equations of motion of any given particle k:

$$\frac{\mathrm{d}\mathbf{V}^{(k)}(t)}{\mathrm{d}t} = \frac{\mathbf{V}^{(k)}(t) - (\mathbf{U}(\mathbf{Y}^{(k)}(t),t) + \mathbf{u}^{(k)})}{\tau_{\mathrm{p}}^{(k)}} - \mathbf{g}$$

and

$$\frac{\mathrm{d}\mathbf{Y}^{(k)}(t)}{\mathrm{d}t} = \mathbf{V}^{(k)}(t),$$

where $\tau_{\mathrm{p}}^{(k)}$ is the particle initial response time, $\mathbf{V}^{(k)}$ are the velocity components, $\mathbf{U}$ is the turbulence airflow field, $\mathbf{Y}^{(k)}$ is the instantaneous location of the particle, and $\mathbf{g}$ is gravitational acceleration. They did so by solving a three-dimensional Navier-Stokes equation and then solving the system of equations iteratively using Gauss-Seidel:

$$u_i^{(k)} = \sum_{m=1;m\neq k}^{N_p} (A^{(mk)} d_i^{(mk)} \mathbf{d}^{(mk)} \cdot \mathbf{u}^{(m)} + B^{(mk)} u_i^{(m)}) + C_i^{(k)}$$

for $i = 1,2,3$ and $k = 1,2,3,...,N_p$, where $u_i^{(k)}$ is the ith component of $\mathbf{u}^{(k)}$, the disturbance flow velocity, $A^{(mk)}$, $B^{(mk)}$, and $C_i^{(k)}$ are ratios of radius and separation distance of the particles, N_p is the number of particles, and $\mathbf{d}^{(mk)}$ is separation distance. The iterative procedure is implemented at the lth iteration as

$$\begin{aligned} u_i^{(k)^{l+1}} = & \sum_{m=1}^{k-1} (A^{(mk)} d_i^{(mk)} \mathbf{d}^{(mk)} \cdot \mathbf{u}^{(m)^{l+1}} + B^{(mk)} u_i^{(m)^{l+1}}) \\ & + \sum_{m=k+1}^{N_p} (A^{(mk)} d_i^{(mk)} \mathbf{d}^{(mk)} \cdot \mathbf{u}^{(m)^{l}} + B^{(mk)} u_i^{(m)^{l}}) + C_i^{(k)}, \end{aligned}$$

for $i = 1,\ 2,\ 3$ and $k = 1,\ 2,\ 3, \ldots, N_p$. The solution is considered to be convergent if

$$\frac{|u_i^{(k)^{l+1}} - u_i^{(k)^{l}}|}{u_{\mathrm{charact}}} \leq \varepsilon,$$

where u_{charact} is considered to be a characteristic velocity and ε is a tolerance parameter. The fourth-order Adams-Bashforth method was used to numerically integrate the two differential equations of motion. Then, the drag force acting on the kth particle can be shown as

$$\mathbf{D}^{(k)}(t) = -6\pi\mu a_k[\mathbf{V}^{(\mathbf{k})}(\mathbf{t}) - (\mathbf{U}(\mathbf{Y}^{(k)}(t),t) + \mathbf{u}^{(k)})], \quad \text{for } k = 1,2,...,N_p,$$

where a_k is radius of the particle and μ is air dynamic viscosity.

E. Shams et al. [13] predicted the motion of bubbles by solving equations of motion in the Lagrangian frame as well. Bubble size variations were modeled by incompressible Rayleigh-Plesset equations solved in a Euler scheme and forces were computed for each bubble. Lagrangian quantities were interpolated to the Eulerian control volumes using

$$\Theta_b(\mathbf{x}_{cv}) = \sum_{b=1}^{N_b} V_b \mathscr{G}_\Delta(\mathbf{x}_{cv}, \mathbf{x}_b),$$

where Θ_b is the local bubble, $\mathbf{x}_{cv}$ is the control volume centroid, N_b is the number of bubbles, V_b is the bubble volume, $\mathscr{G}_\Delta$ is the interpolation function, and $\mathbf{x}_b$ is the bubble position. Once all forces were known for each bubble, position and velocity were advanced using an explicit Euler scheme from $t^{n+1/2}$ to $t^{n+3/2}$. Gaussian interpolation is given by

$$\mathscr{G}_\sigma(\mathbf{x}, \mathbf{x}_b) = \frac{1}{\sigma\sqrt{(2\pi)^3}} \exp\left[-\frac{\sum_{k=1}^{3}(x_k - x_{b,k})^2}{2\sigma^2} \right],$$

where σ is the kernel width and $x_k, x_{b,k}$ denote the available data point on the grid and location of bubble, respectively. Then, the net force acting on each bubble is shown as

$$\mathbf{F}_{\ell \to b} = \mathbf{F}_G + \mathbf{F}_P + \mathbf{F}_D + \mathbf{F}_L + \mathbf{F}_{AM} + \mathbf{F}_{coll} + \mathbf{F}_{\dot{R}_b}$$

where drag force and lift force are

$$\mathbf{F}_D = -\frac{1}{2} C_D \rho_\ell \pi R_b^2 |\mathbf{u}_b - \mathbf{u}_\ell| (\mathbf{u}_b - \mathbf{u}_\ell)$$

and

$$\mathbf{F}_L = -C_L \rho_\ell V_b (\mathbf{u}_b - \mathbf{u}_\ell) \times \nabla \times \mathbf{u}_\ell,$$

respectively. Here, $\mathbf{F}_G$ is gravitational force, $\mathbf{F}_P$ is pressure force, $\mathbf{F}_{AM}$ is added mass force, $\mathbf{F}_{coll}$ is the bubble collision force, $\mathbf{F}_{\dot{R}_b}$ is the momentum transfer, C_D is the drag coefficient,ρ_ℓ is the liquid density, R_b is the bubble radius, $\mathbf{u}_b$ is the bubble velocity in the disperse phase, $\mathbf{u}_\ell$ is the bubble velocity in the fluid phase, and C_L is the lift coefficient.

S.V. Apte et al. [2] used a formulation that consisted of Eulerian fluid and Lagrangian particle equations. They accounted for the displacement of the fluid caused by the particles,

as well as the interphase transfer. The drag force caused by the particle motion through gas was defined as

$$D_p = \frac{3}{8} C_d \frac{\rho_f}{\rho_p} \frac{|\mathbf{u}_{f,p} - \mathbf{u}_p|}{R_p},$$

where R_p is the radius of each particle, ρ_p and ρ_f are the particle and fluid density, respectively, and $\mathbf{u}$ are the velocity components. The drag coefficient C_d was given by

$$\begin{aligned} C_d &= \frac{24}{Re_p}(1 + aRe_p^b)\Theta_f^{-2.65}, \text{ for } Re_p < 1000 \\ &= 0.44\Theta_f^{-1.8}, \text{ for } Re_p \geq 1000, \end{aligned}$$

where Re_p is the particle Reynolds number, Θ_f is the volume fraction, and a and b are constants 0.5 and 0.687, respectively.

2.2 Previous Work on Drag Forces Acting on Nonspherical Particles

Next, we take a look at previous methods used to calculate the drag forces on various nonspherical particles. D. Leith [10] considered a dynamic shape factor, K, along with Stokes' law to predict drag forces on nonspherical particles. This shape factor is defined for an arbitrary shape such that the product of K and Stokes' original drag law for a sphere with identical volume as the arbitrary shape gives an accurate drag force:

$$\begin{aligned} F_D &= 3\pi\mu V \left[\left(\frac{1}{3} \right) d_n + \left(\frac{2}{3} \right) d_s \right] \\ &= 3\pi\mu V d_n \left[\frac{1}{3} + \frac{2}{3} \left(\frac{d_s}{d_n} \right) \right] \\ &= 3\pi\mu V d_n K, \end{aligned}$$

where d_n is the diameter of a sphere with identical projected area as the arbitrary particle projected normal to its direction of motion, and d_s is the diameter of a sphere with identical surface area as the arbitrary particle. We can see that the dynamic shape factor K is therefore defined as:

$$K = \left[\frac{1}{3} + \frac{2}{3} \left(\frac{d_s}{d_n} \right) \right].$$

While this shape factor considered by Leith does depend on particle orientation, the theory only predicts shape factors for shapes whose boundaries can be easily described in a simple coordinate system, such as prisms, spheroids, needles, toroids, etc. Therefore, our drag force theory must extend to more irregular shapes that are more commonly found in nature, such as limpet shells and their fragments.

Dioguardi et. al. [7] developed a new drag force theory for irregulary shaped particles with a large range of Reynolds numbers. They considered 143 different volcanic ash and lapilli particles and used a shape factor Ψ that only considered each particle's circularity X and sphericity Φ [5]:

$$\Psi = \frac{\Phi}{X}, \tag{2.1}$$

where

$$\Phi = \frac{A_{sph}}{A_p}, \quad \text{and} \quad X = \frac{P_{mp}}{P_c}.$$

Here, A_{sph} is the surface area of a sphere equivalent to the irregular particle by means of diameter of the volume of said equivalent sphere d_{sph}:

$$d_{sph} = \left(\frac{6m}{\pi\rho_p}\right)^{1/3},$$

where m and ρ_p are the mass and density of the particle. A_p is the irregular particles's surface area. P_{mp} and P_c are the maximum projection perimeter and the perimeter of the circle equivalent to the maximum projection area of the particle. Using a reference law for spheres, they obtained a fitting law that best interpolated measured points, resulting in a new drag force theory:

$$C_d = \frac{24}{Re}\left(\frac{1-\Psi}{Re}+1\right)^{0.25} + \frac{24}{Re}\left(0.1806Re^{0.6459}\right)\Psi^{-(Re^{0.08})} + \frac{0.4251}{1+\frac{6880.95}{Re}\Psi^{5.05}}.$$

They considered Reynolds numbers ranging from 0.03 to 10,000 and particle shape factors Ψ ranging from 0.335 to 0.943. Orientation of the irregular particles was not considered. Measured versus calculated drag coefficient results using this new drag force theory are shown in Figure 2.1. While Dioguardi et al.'s method did consider Reynolds number and shape factor, it did not consider orientation of particles.

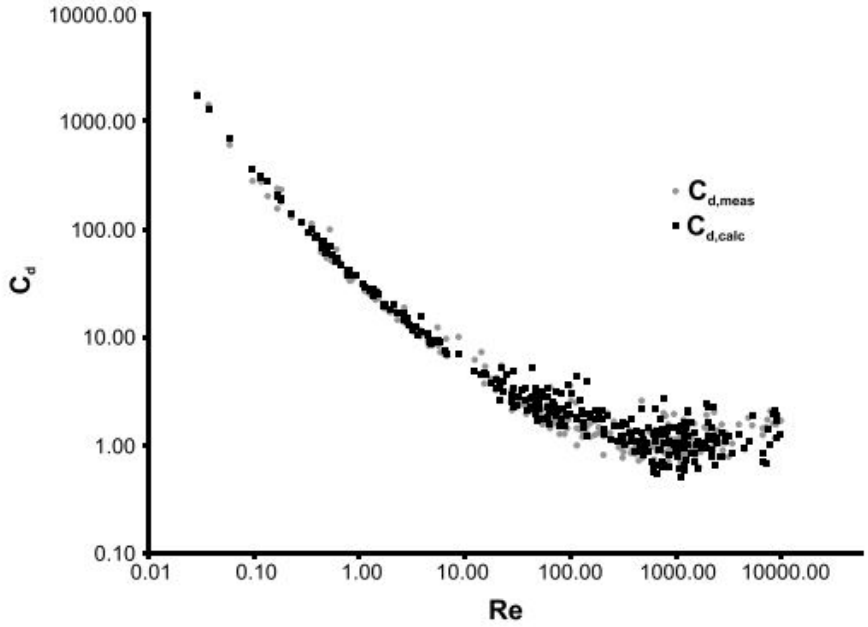

Figure 2.1: Measured vs. Calculated Drag Coefficients using Dioguardi Drag Force Theory Drag coefficients C_d versus Reynolds number comparing measured aerodynamic drag force on 143 different irregularly shaped particles. Black squares denote the calcuated drag coefficients from Dioguardi's new drag force law while grey dots denote drag coefficients measured in experiments.

K. A. Ahmad [1] used various $k-\varepsilon$ and $k-\omega$ turbulence models in CFD simulations to predict the flow field around a NACA 0012 airfoil with various pitch orienations. For our research purposes, we consider the NACA 0012 airfoil as a symmetrical particle with respect to the direction of flow. A comparison of Ahmad's simulations with experimental data suggest the best model to use for the case of a symmetrical particle with various pitch orientations is the SST $k-\omega$ turbulence model with resolved boundary layers from a $O(y^+ = 1)$ grid size. The meaning and importance of y^+ will be discussed in Section 4.2.1. Results for Ahmad's experimental and simulation results are shown in Figure 2.2.

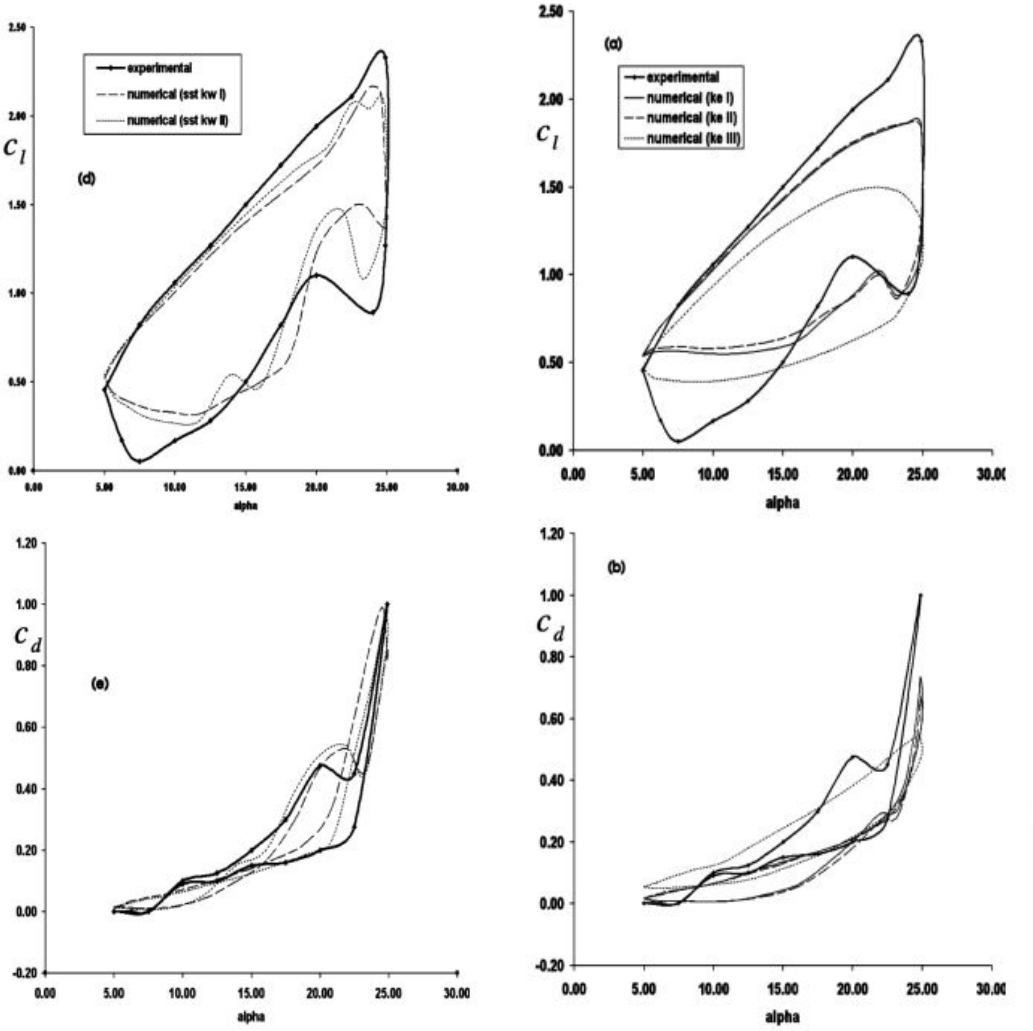

Figure 2.2: Hysterisis Loops of Pitching Airfoil using SST $k-\omega$ Model vs. $k-\varepsilon$ Model Hysterisis loops of Ahmad's experimental results, **(d,e)** simulation results using the SST $k-\omega$ model with standard wall functions (SST $k\omega$ I), simulation results using the SST $k-\omega$ model with a two-layer near wall model (SST $k\omega$ II), **(a,b)** simulation results using the $k-\varepsilon$ model with standard wall functions ($k\varepsilon$ I), simulation results using the $k-\varepsilon$ model with non-equilibrium wall functions ($k\varepsilon$ II), and finally, simulation results using the $k-\varepsilon$ model with a two-layer near wall model ($k\varepsilon$ III). C_l and C_d are reported lift and drag coefficients, respectively.

2.3 Previous Work on Drag and Lift Forces Acting on Limpet Shells

More specific to geometries that we consider in this dissertation, we examine laboratory experiments performed by M. Denny in 1989 [6] on three different limpet shells. They used a transducer for both drag and lift experiments and attached the shells by a bolt to a force platform, with the shell lying flush with the surface of the flume's working section. A diagram of Denny's lab experiments is shown in Figure 2.3.

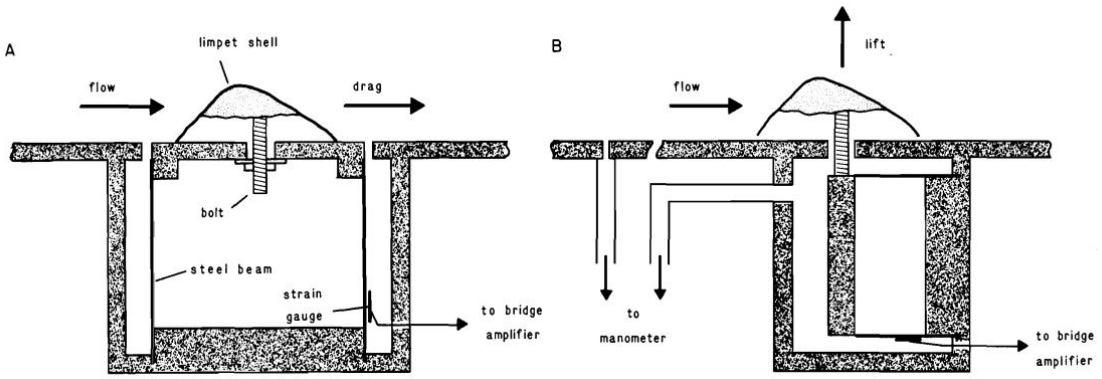

Figure 2.3: Diagram of Denny's Experiment on Limpet Shells
(a) The drag transducer. **(b)** The lift transducer.

Denny considers the following equation for their drag coefficient C_d:

$$C_d = 2F_d/(\rho u^2 A_p), \tag{2.2}$$

where A_p is the area of the shell projected in the direction of flow. In Denny's experiments they only considered three different orientations: anterior of the shell facing the inlet, posterior of the shell facing the inlet, and the side of the shell facing the inlet. For our purposes, this is equivalent to three orientations of 0°, 90°, and 180°, respectively, on the z-axis, all else held constant. Denny also only considered Reynolds numbers ranging from 10^4 to 10^6.

The area considered by Denny's drag force coefficient in equation (2.2) is strictly dependent on orientation of the shell, since the projected area of the shell relative to the flow will change greatly for the 90° yaw and slightly between the 0° and 180° yaw. Because Denny's drag force model is therefore orientation-dependent in the object frame of reference, their model cannot predict the hydrodynamic forces for a constant object at various orientations. Therefore, we will have to consider another type of area in our new drag force model.

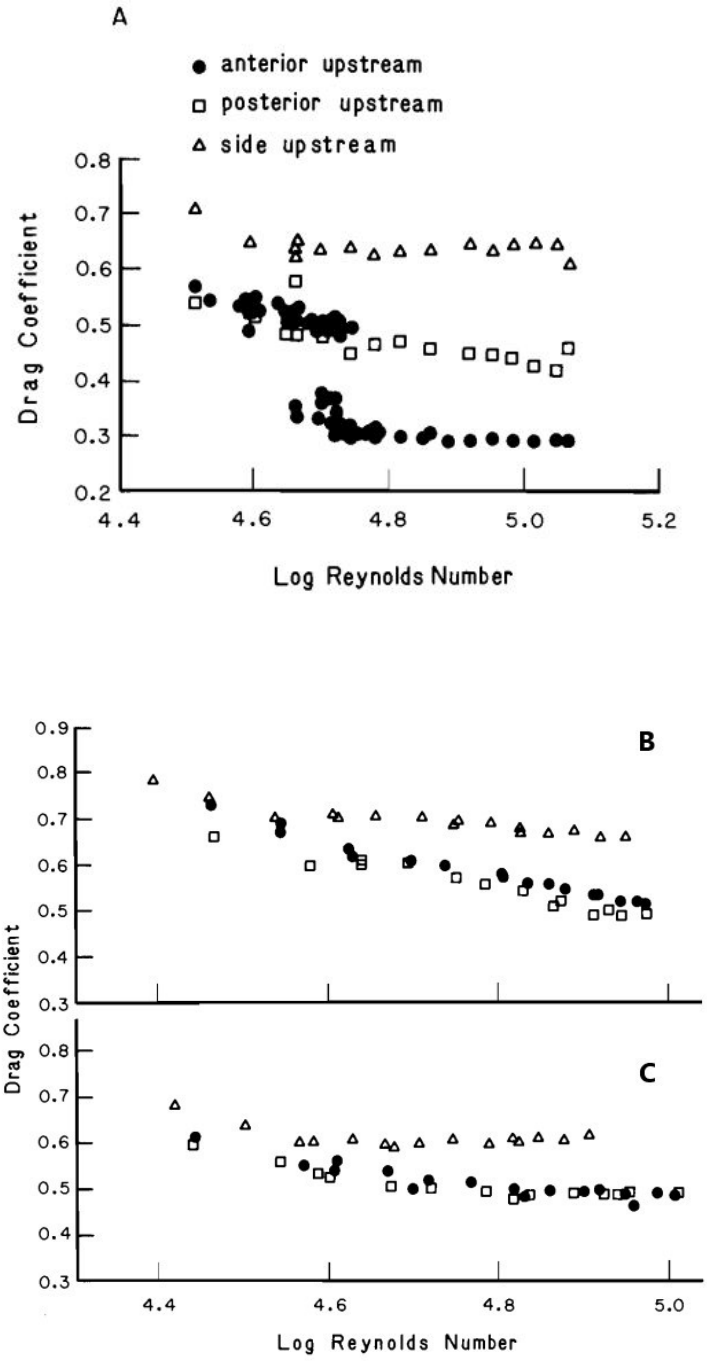

Figure 2.4: Drag Coefficient Results for Three of Denny's Limpet Shells
Drag coefficient results versus Reynolds number from Denny's lab experiments for shells A, B, and C. Dots denote the orientation $(0\ 0\ 0)^\circ$. Squares denote the orientation $(0\ 0\ 180)^\circ$. Triangles denote the orientation $(0\ 0\ 90)^\circ$.

2.4 Previous Work on Drag and Lift Aerodynamics of Flat Plates with Varying Aspect Ratios and Varying Pitch Angles

Next, we take a look at a method used to calculate the aerodynamic forces acting on an object with varying angles of attack. X. Ortiz et al. [11] performed wind tunnel experiments on flat plates with low aspect ratios at low Reynolds numbers. The purpose of their experiments was to better understand the wind loads on photovoltaic modules installed on roofs with high angles of inclination, as well as better understand the aerodynamic forces of turbine blades with typical angles of attack up to 90°. They therefore focus on a drag, C_D, and lift, C_L, coefficient model that depends on orientation, specifically angle of attack:

$$C_L = \sin\alpha\cos\alpha\Big(K_P\cos\alpha + \pi\sin\alpha\Big) \tag{2.3}$$

and

$$C_D = C_{d0} K C_L^2, \tag{2.4}$$

where C_{d0} is the zero-angle drag due to viscosity, K_P and K are a potential component for lift and an induced drag factor, respectively, and α is angle of attack.

At low angles of attack, the viscosity greatly influences the aerodynamic drag, with C_{d0} typically being around 0.015. As angle of attack increases, however, pressure then begins to influence the aerodynamic drag even more, and eventually

$$C_D/C_L = \tan\alpha, \tag{2.5}$$

making equations (2.2) and (2.3) invalid. If we let α_m be the highest angle for which equations (2.2) and (2.3) are valid, then α_m is also a lower bound for which equation (2.4) is valid and all aerodynamic forces acting on a plate are greatly influenced by pressure acting in the direction normal to the plate's surface, rather than viscosity. Ortiz et al. observed that α_m is strongly dependent on the aspect ratio of the plate and equation (2.4) is valid for an increasing range of angles of attack as aspect ratio increases.

AR	α_m (°)	K_p	K
0.5	35	0.831	0.67
0.75	33	1.26	0.565
1.0	28	1.59	0.53
1.25	20	1.85	0.483
1.5	15	2.10	0.417
1.75	14	2.35	0.409
2.0	13	2.59	0.374

Figure 2.5: Parameters for Flat Plate Drag and Lift
Parameters for aspect ratio, AR, highest angle allowed for equations (2.2),(2.3), α_m, and lift and drag factors, K_P and K.

Chapter 3

A NEW MODEL FOR PREDICTING DRAG AND LIFT FORCES ON ARBITRARILY SHAPED OBJECTS

3.1 The Shear Stress Transport $k-\omega$ Turbulence Model

The shear stress transport (SST) $k-\omega$ turbulence model is a commonly used turbulence model in the family of Reynolds-averaged Navier-Stokes (RANS) turbulence models, which model all effects of turbulence. The time-dependent mean velocity $\mathscr{U}$ is solved by the RANS equation

$$\frac{\partial \mathscr{U}_i}{\partial t} + \frac{\partial \mathscr{U}_i \mathscr{U}_j}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\nu \partial^2 \mathscr{U}_i}{\partial x_j^2} - \frac{\partial \overline{u_i' u_j'}}{\partial x_j} \tag{3.1}$$

and the conservation of mass is enforced by

$$\frac{\partial \mathscr{U}_i}{\partial x_i} = 0. \tag{3.2}$$

In equation (3.1), the Reynolds stress is usually modeled as

$$\overline{u_i' u_j'} = -\nu_T \frac{\partial \mathscr{U}_i}{\partial x_j}, \tag{3.3}$$

where the eddy viscosity ν_T can be modeled by the time-dependent PDEs for turbulent kinematic energy k and specific dissipation rate ω. The SST $k-\omega$ turbulence model solves these two separate PDEs, or transport equations, that account for effects like diffusion of turbulent energy and convection.

The standard $k-\omega$ turbulence model is designed for low Reynolds numbers, as it works best for flow where the boundary layer is relatively thick, allowing the viscous sublayer to be easily resolved [4]. Thus, it is best used for near-wall treatment in simulations. The SST $k-\omega$ model, however, uses model behavior from the $k-\varepsilon$ turbulence model in the free-stream, avoiding sensitivity at the inlet usually observed by the standard $k-\omega$ model. The $k-\varepsilon$ model is most reliable for free-shear flows, where the flow develops a mean velocity gradient in the absence of boundaries, so this model would not have worked well for us by itself. The SST $k-\omega$ model uses blending functions F_1 and F_2 to switch from

regular $k-\omega$ behavior to $k-\varepsilon$ behavior in the free-stream. The time-dependent PDE representations of k and ω are

$$\frac{\partial k}{\partial t}+u_j\frac{\partial k}{\partial x_j}=P_k-\beta^* k\omega+\frac{\partial}{\partial x_j}\left[(\nu+\sigma_k\nu_T)\frac{\partial k}{\partial x_j}\right] \tag{3.4}$$

and

$$\frac{\partial \omega}{\partial t}+u_j\frac{\partial \omega}{\partial x_j}=\alpha S^2-\beta\omega^2+\frac{\partial}{\partial x_j}\left[(\nu+\sigma_\omega\nu_T)\frac{\partial \omega}{\partial x_j}\right]+2(1-F_1)\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i}, \tag{3.5}$$

with a blending function

$$F_1=\tanh\left\{\left\{\min\left[\max\left(\frac{\sqrt{k}}{\beta^*\omega y},\frac{500\nu}{y^2\omega}\right),\frac{4\sigma_{\omega 2}k}{CD_{k\omega}y^2}\right]\right\}^4\right\},$$

where $F_1=1$ inside the boundary layer and $F_1=0$ in the free-stream. $CD_{k\omega}$ is a function defined as

$$CD_{k\omega}=\max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i},10^{-10}\right).$$

Kinematic eddy viscosity can be solved by

$$\nu_T=\frac{a_1 k}{\max(a_1\omega,SF_2)}. \tag{3.6}$$

Here, F_2 is another blending function defined as

$$F_2=\tanh\left[\left[\max\left(\frac{2\sqrt{k}}{\beta^*\omega y},\frac{500\nu}{y^2\omega}\right)\right]^2\right].$$

The model also considers a production limiter P_k to alleviate spurious production of k in stagnation regions:

$$P_k=\min\left(\tau_{ij}\frac{\partial u_i}{\partial x_j},10\beta^* k\omega\right).$$

x_i and x_j are spatial components, u_i and u_j are velocity components, and y is a distance. Table 3.1 defines some constants used:

Table 3.1: *Constant Definitions*.

β	0.075
β^*	0.090
σ_ω	0.500
$\sigma_{\omega 2}$	0.856

In OpenFOAM, we use equations

$$k = \frac{3}{2}(UI)^2 \tag{3.7}$$

and

$$\omega = \frac{k^{0.5}}{0.55T} \tag{3.8}$$

to calculate good initial guesses for k and ω as boundary conditions [14], and then OpenFOAM uses equation (3.1) to compute k, ω, pressure p, kinematic viscosity ν_t, and each component of velocity at each time-step, thereby calculating the drag and lift coefficients we need. For equations (3.7) and (3.8), U is the velocity of flow at the top of the channel, or the free-stream, I is the turbulence intensity, and T is a turbulence length scale.

3.1.1 Numerical Schemes Utilized by OpenFOAM

OpenFOAM uses different numerical schemes to solve the PDEs mentioned above. Specifically, a divergence scheme describes the net rate that a property changes as a function of space, a time scheme defines how a property is integrated as a function of time, and a gradient scheme of a property ϕ is represented as

$$\nabla\phi = \mathbf{s}_1\frac{\partial}{\partial x_1}\phi + \mathbf{s}_2\frac{\partial}{\partial x_2}\phi + \mathbf{s}_3\frac{\partial}{\partial x_3}\phi,$$

where $\mathbf{s}$ represents the unit vectors of the 3-dimensional space. For our time scheme we choose a steady-state scheme, which does not solve for the time derivatives. For our divergence scheme we choose a bounded Gaussian linear upwind scheme. For our gradient scheme we choose a Gaussian linear scheme.

A Gaussian scheme establishes the standard finite volume discretization of Gaussian integration, while a Gaussian linear scheme further establishes linear interpolation or central differencing. The bounded variant of these schemes correspond to the treatment of the time derivative for convection, e:

$$\frac{De}{Dt} = \frac{\partial e}{\partial t} + \mathbf{U}\cdot\nabla e = \frac{\partial e}{\partial t} + \nabla\cdot(\mathbf{U}e) - (\nabla\cdot\mathbf{U})e. \tag{3.9}$$

For the numerical solutions of incompressible flows, $\nabla\cdot\mathbf{U} = 0$ at convergence. Then, the third term of equation (3.9) is also zero. Before convergence, $\nabla\cdot\mathbf{U} \neq 0$, and the third term of equation (3.9) aids in maintaining boundedness of the solution variable, promoting better convergence. Choosing a bounded variant of Gaussian linear schemes automatically discretizes the third term of equation (3.9) with the advection term. The upwind variant of our divergence scheme is simply a first-order bounded variant.

3.2 Transforming the Fluid-Particle Drag Force Equation

In the previous section, we consider the RANS equations used by OpenFOAM. Once these solutions are obtained, OpenFOAM calculates the hydrodynamic forces for drag and lift by integrating the computed pressure and viscous stress. We use the fluid-particle drag force from the Lagrangian method for tracking particle movement in the computational frame of reference to develop our predictive model for the forces. For example, the drag force is

$$F_D = \frac{1}{2} u_1 \rho C_D A ||u||, \tag{3.10}$$

where ρ is fluid density, C_D is the drag force coefficient that is the function

$$C_D(\alpha, \delta, \gamma, Re, c, a, b),$$

A is the particle surface area, u_1 is the mainstream velocity, and α, δ, and γ are orientations on each axis, respectively. c, a, and b are parameters of a shape factor that will be discussed in Chapter 6. In order to develop a new model for predicting the hydrodynamic forces acting upon an irregular object, we must simplify the force model by eliminating the orientation dependence from the force coefficients. We do so by defining a rotation matrix for each orientation of our limpet shell and "rotating" equation (3.10), allowing us to work with the equation in the object frame of reference. Our rotation matrices for each orientation of our shells are defined below:

$$R_z(k) = \begin{bmatrix} \cos(\alpha_k) & \sin(\alpha_k) & 0 \\ -\sin(\alpha_k) & \cos(\alpha_k) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_y(k) = \begin{bmatrix} \cos(\delta_k) & 0 & \sin(\delta_k) \\ 0 & 1 & 0 \\ -\sin(\delta_k) & 0 & \cos(\delta_k) \end{bmatrix}$$

$$R_x(k) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma_k) & -\sin(\gamma_k) \\ 0 & \sin(\gamma_k) & \cos(\gamma_k) \end{bmatrix}$$

Then, we have a unique rotation matrix $R(k)$ for each orientation of our shells:

$$R(k) = R_z(k) \cdot R_y(k) \cdot R_x(k). \tag{3.11}$$

We use this unique rotation matrix to convert our simulated forces and our velocity components from the computational frame of reference to the object frame of reference:

$$\hat{F}_i(k) = R_{ij}^T(k) F_j, \tag{3.12}$$

and

$$\hat{u}_i(k) = R_{i1}^T(k)u_1, \tag{3.13}$$

where $i = 1$ for the longitudinal components and $i = 2$ for the normal components in the object frame. Then, we hypothesize that we can obtain a linear regression for the forces in the object frame of reference with

$$\hat{F}_i = \frac{1}{2}\hat{u}_j \rho A C_{ij} ||u||, \tag{3.14}$$

where C_{ij} is now a function $C_{ij}(Re, c, a, b)$ and is now orientation-independent in the object frame. Here, "^" denotes the object frame of reference.

In MATLAB, this procedure looks like the following: we begin with our initially positioned shell in the computational frame of reference, where the longitudinal velocity spans from the anterior to the posterior of the shell, and the normal velocity spans from the bottom of the shell and upwards, or the concave to convex side. Then, if the longitudinal velocity is pointing to the right and normal velocity is pointing up, then spanwise velocity is pointing outwards towards the reader. This is our reference point for the rotation matrices in the object frame. For each orientation, we compute the drag and lift forces F_D and F_L using our simulated OpenFOAM drag and lift coefficients C_D and C_L, normalized by density. Next, we rotate the simulated forces into the object frame. Then, we rotate our velocity to obtain our component-wise velocities in the object frame, where $u_2 = u_3 = 0$ since flow is only moving in one direction. Finally, we regress our coefficients in the object frame of reference using the backslash operator.

3.3 Defining Area for our New Force Model

Unlike Denny's experiements observed in Chapter 2, we must consider an area in our drag force model that is not orientation-dependent. Therefore, we instead consider the total surface area of our limpet shell. This of course will change with different shells and shell fragments, but area will still be orientation-independent. We use Paraview to calculate the total surface area of our limpet shell geometry and use this value as A in our drag force model (3.10).

For purposes of validating our drag coefficient results against Denny's, we still must compute the projected surface area of our shells with orientations $(0\ \ 0\ \ 0)^\circ$, $(0\ \ 0\ \ 90)^\circ$, and $(0\ \ 0\ \ 180)^\circ$. Note that we define our initial orientation $(0\ \ 0\ \ 0)^\circ$ as the case of the shell concave side down with the anterior facing the inlet. Once we have the projected surface areas, we can scale our hydrodynamic force coefficients by a ratio of the area we use and the area Denny uses. To do so, we use MATLAB to slice the geometry of our shell on the

z-axis and project the cells of what is left on the xy-axis. Then, we perform a Delaunay triangulation of the cells and calculate the area of each triangle. Computing the sums of the areas of all triangles gives us our projected surface areas.

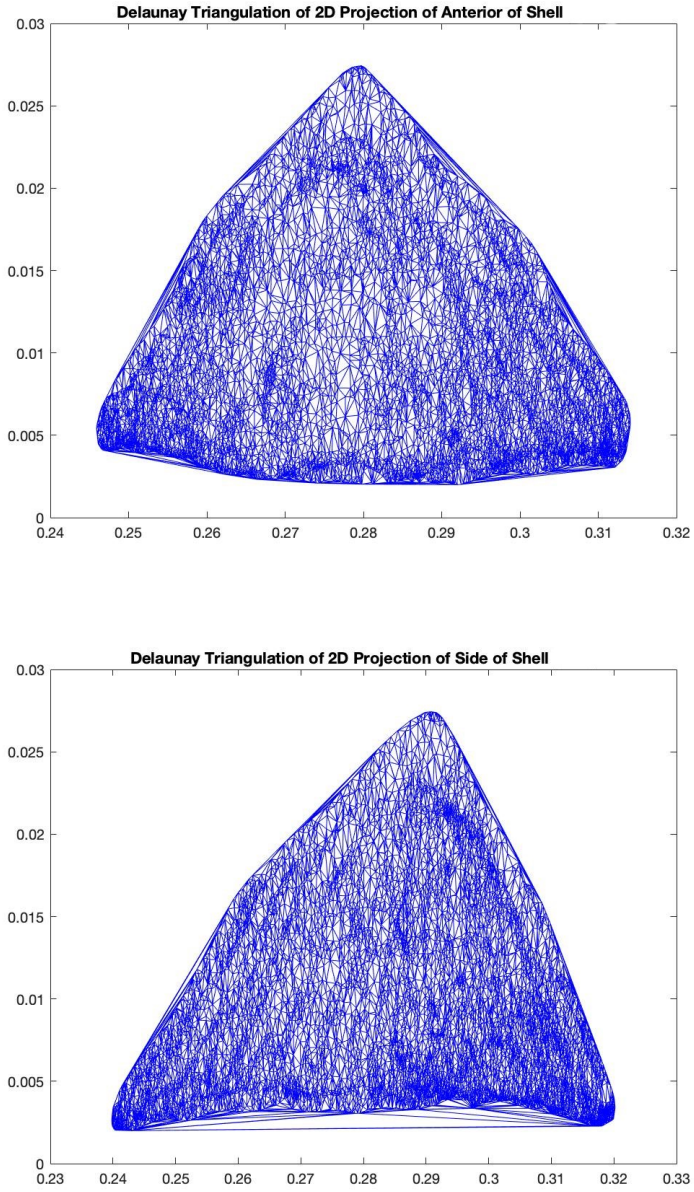

Figure 3.1: Delaunay Triangulation of Shell Geometry and Denny's Orientations **(top)** Projected surface of the anterior inlet-facing shell was calculated to be $1.1 \times 10^{-3} m^2$, resulting in a scaling factor of 9.68 we will use to validate our results against Denny's. **(bottom)** Projected surface of the side inlet-facing shell was calculated to be $1.2 \times 10^{-3} m^2$, resulting in a scaling factor of 8.75 we will use to validate our results against Denny's. For simplicity, we consider the same scaling factor for the posterior inlet-facing shell as the anterior inlet-facing shell.

Chapter 4

COMPARATIVE STUDY - USING THE SST $k-\omega$ TURBULENCE MODEL ON A FLAT PLATE WITH VARYING ORIENTATIONS

4.1 Introduction

In order to validate our use of the SST $k-\omega$ turbulence model in OpenFOAM, we consider the same air tunnel experiments discussed in Section 2.4, conducted by Xavier Ortiz, David Rival, and David Wood at the University of Calgary and Queen's University, Canada in 2015. We replicate one case of the plates at 12 different angles of attack using the same SST $k-\omega$ turbulence model validated by Ahmad with a grid size of $O(y^+ = 1)$, which will be further explained in Section 4.2.1. We then compare our calculated drag and lift coefficient results with those reported in Ortiz, et al., as well as our predicted and reported hydrodynamic forces based off our new model to validate our theory.

4.2 Obtaining and Improving a Plate Geometry

We first create a stereolithography file of a rectangular plate in Blender and scale the dimensions to match those of the plate with an aspect ratio of 6 in the Ortiz, et al. paper. Our resulting plate has a length of 0.066 meters, a width of 0.4 meters, and a thickness of 0.002 meters. Next, we improved the surface mesh of the geometry by subdividing elongated triangles using a combination of Meshmixer and Blender. We rounded and smoothed the edges and corners of the plate to allow better surface layer addition in our domain meshing steps.

4.2.1 Explicitly Resolving Boundary Layers

Wall functions are empirically derived for the purpose of satisfying the physics in the near-wall regions. While they are commonly used, they can sometimes skew results at these near-wall regions, especially for complex flows. Therefore, resolving our boundary layers within the mesh instead is the more accurate route [12]. To resolve our boundary layers in our simulations and avoid the use of wall functions in our boundary conditions, we must choose a y^+ value, a non-dimensional distance, that determines the proper size of cells in near-wall regions. Specifically, as proven by Ahmad and discussed in Section 2.2, we desire

an $O(y^+ = 1)$ grid size, as this will allow us to integrate the turbulence to the wall in a low-Reynolds turbulence model, such as SST $k-\omega$. Although we do consider some higher Reynolds numbers, flow is naturally low-Reynolds at the viscous sublayers, so this will work for our applications as well. The first step to determining the Δx of $O(y^+ = 1)$ grid size, the cell size we desire in our boundary regions, is explicitly solving for our mainstream velocity u:

$$u = \frac{Re * \nu}{L} = \frac{8.27 \times 10^4 * 1.5 \times 10^{-5}}{0.066} = 18.79 m/s, \tag{4.1}$$

where ν is the kinematic viscosity of air, L is the shortest length of the plate, and Re is the Reynolds number, given by Table 2 in Ortiz et al. Next, we must implicitly compute the shear velocity, or friction velocity, u_* by use of a combination of the secant method and bisection on the following equation:

$$u = 2.5 u_* ln\left(9y\frac{u_*}{\nu}\right) \longrightarrow u_* = 0.05 m/s, \tag{4.2}$$

where y is the height of the center of the plate. Now we can compute our needed grid size Δx of $O(y^+ = 1)$:

$$\Delta x = \frac{\nu}{u_*} = \frac{1.5 \times 10^{-5}}{0.05} = 0.0003 m. \tag{4.3}$$

Finally, we coarsen the surface mesh so the smallest cell size is approximately Δx, or 0.0003 meters, allowing us to resolve our boundary layers using a grid size of $O(y^+ = 1)$. In Figure 4.1, we demonstrate our final triangulated surface mesh of our plate in simulation with an aspect ratio of 6, as well as a close -up of the triangular cells with the smallest cell size being approximately $\Delta x = 0.0003 m$.

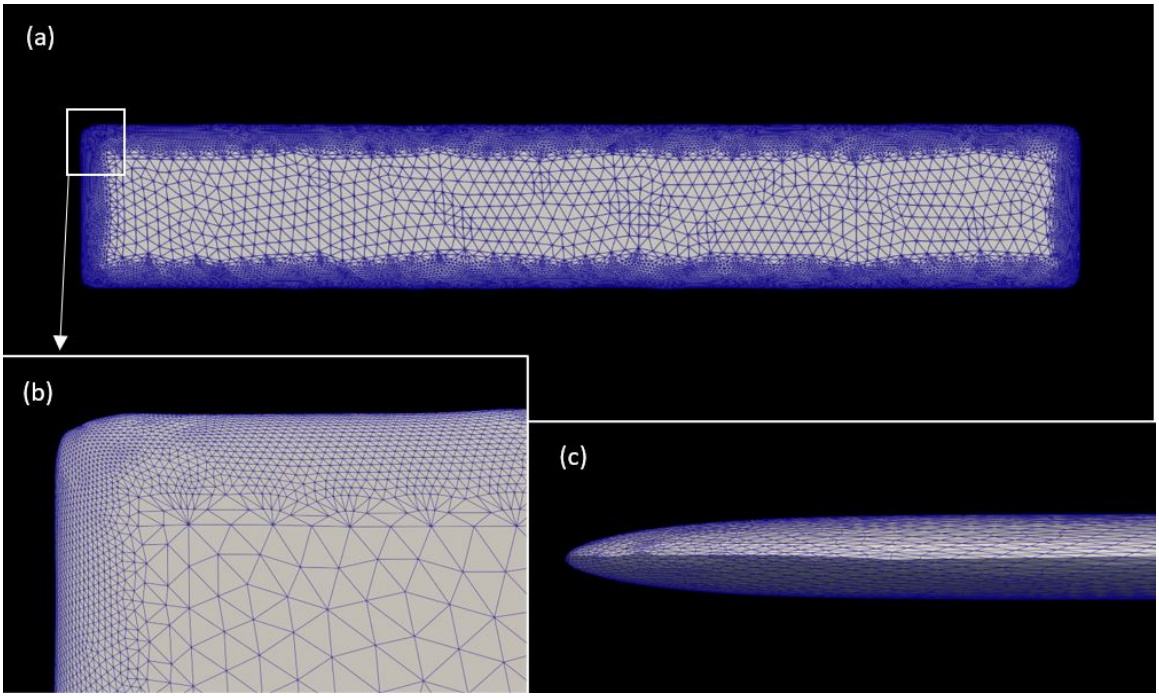

Figure 4.1: Triangulated Plate Geometry
(a) Our final triangulated surface mesh of a plate with an aspect ratio of 6. **(b)** Close-up of triangles with a smallest cell size of $\Delta x = 0.0003m$. **(c)** Beveled edges of our plate.

4.3 Setting up blockMesh and snappyHexMesh Dictionaries

We use OpenFOAM's mesh generators blockMesh and snappyHexMesh to create our domain meshes. The first, blockMesh, builds a simple mesh out of blocks, while snappyHexMesh takes the blockMesh and essentially chisels it into the mesh we desire. We create a blockMesh domain with dimensions $1.2 \times 1.6 \times 0.6$ meters and using snappyHexMesh we include 6 refinement levels on the floor and wake of the plate. Starting with a cell size of 0.0384 meters, we refine until our sixth refinement level is 0.0006 meters. Then, we add two surface layers to our plate geometry with a final layer thickness of 0.5 and an expansion ratio of 1.1, resulting in an overall thickness of 0.000286 meters. This is very close to our desired Δx of 0.0003 meters, ensuring that not only will our surface mesh resolve the boundary layers, but now also the mesh inside the domain will as well. We set our plate 0.15 meters away from the inlet and the center of the plate $\frac{3}{4}b + 10\Delta x$ above the floor, as stated in the Ortiz, et al. paper. Here, b is the length of the plate, or 0.066 meters. We add the additional height of $10\Delta x$ to allow for adding surface layers on the 90 degree angle of attack case where the geometry will be closest to the floor.

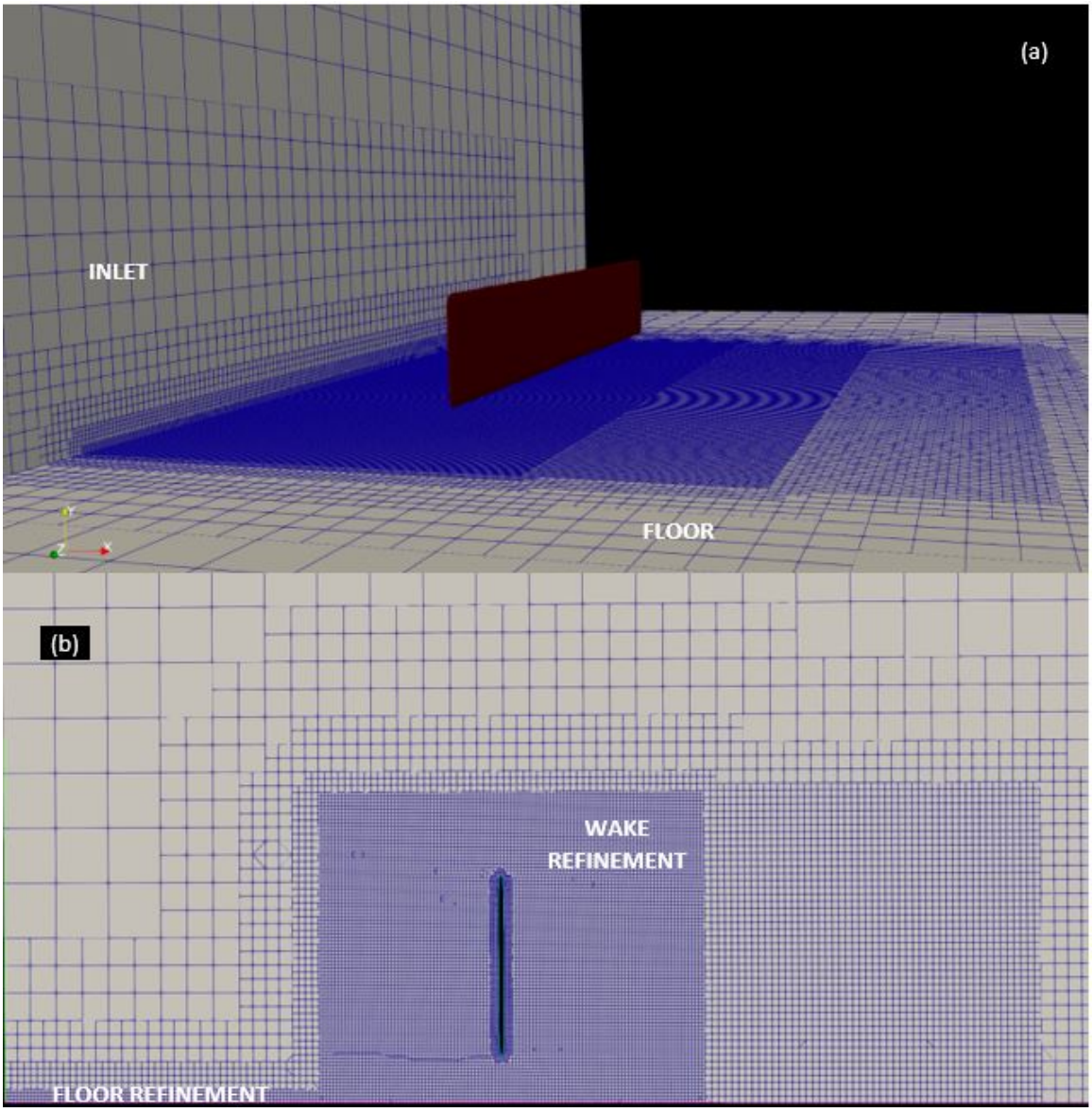

Figure 4.2: Plate with Angle of Attack $\alpha = 90^\circ$ in Simulation
(a) Our replication of Ortiz's experiement for an angle of attack $\alpha = 90^\circ$ and plate aspect ratio of 6. **(b)** Slice of our mesh through the z-axis displaying our wake and floor refinement levels around the plate.

4.4 Setting up Boundary and Initial Conditions

We calculate our initial conditions using equations provided by the SST $k-\omega$ turbulence model properties:

Table 4.1: *Initial Conditions for Plate Simulations with* $AR = 6$ *and* $Re = 8.27 \times 10^4$.

Top channel velocity U	$U = 2.5u_* ln\left(9H\frac{u_*}{\nu}\right)$	$U = 1.225\ m/s$
Turbulence length scale T	$T = 0.4H$	$T = 0.24$
Channel Reynolds number Re_C	$Re_C = \frac{HU}{\nu}$	$Re_C = 49,000$
Intensity I	$I = T(Re_C^{-0.125})$	$I = 6.22\%$
Turbulence kinetic energy k	$k = 1.5(UI)^2$	$k = 0.00964\ m^2/s^2$
Turbulence specific dissipation rate ω	$\omega = \frac{k^{0.5}}{0.55T}$	$\omega = 0.74694\ s^{-1}$

Here, H is the height of the domain. Next, we experiment with a few different boundary conditions for k, ω, and ν_t, where ν_t is turbulence kinematic viscosity. Because we are resolving our boundary layers with our grid step size Δx, wall functions should not be necessary in our simulations, as their purpose is to also resolve these boundary layers. We wish to observe convergence of our drag force coefficient using wall functions in our boundary conditions versus just a wall function for ω versus a fixed value of nearly infinity for ω and zero for ν_t and k. The purpose of this experiment is to determine that we have in fact resolved our boundary layers with Δx. In Figure 4.3, we observe convergence of the drag coefficient on our plate simulation using wall functions, fixed values, or a combination of both on the walls, the floor, and the plate, for ω, ν_t, and k.

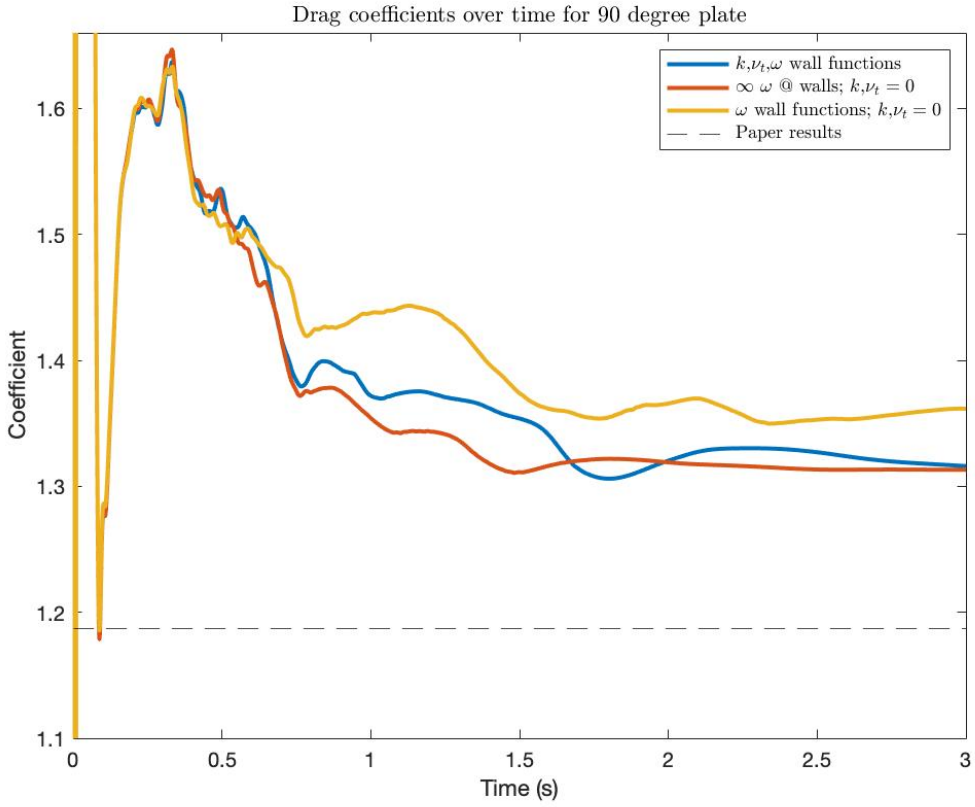

Figure 4.3: Convergence of Drag Coefficient with Various Boundary Conditions
The blue line represents convergence of the drag coefficient using wall functions for ω, k, and ν_t, the red line represents convergence of the drag coefficient using a fixed value of ∞ for ω and a fixed value of 0 for k and ν_t, and the yellow line represents convergence of the drag coefficient using a wall function for ω and a fixed value of 0 for k and ν_t.

We determine that fixed values of our boundary conditions on walls for ω, ν_t, and k provide the best convergence as well as a solution closest to the results provided by Ortiz. Therefore, our boundary layers appear sufficiently resolved and we do not have to further reduce the grid size of the mesh layers adjacent to no-slip boundaries.

For these simulations, we applied the following boundary conditions listed in Table 4.2:

Table 4.2: *Boundary Conditions for Plate Simulations with* $AR = 6$ *and* $Re = 8.27 \times 10^4$.

Parameter	Floor	Top Wall	Side Walls	Plate	Inlet
U	`noSlip`	`slip`	`slip`	`fixedValue (0 0 0)`	`fixedProfile`
k	`fixedValue 0`	`slip`	`slip`	`fixedValue 0`	`fixedValue` from Table 3.1
ω	`fixedValue` ∞	`slip`	`slip`	`fixedValue` ∞	`fixedValue` from Table 3.1
ν_t	`fixedValue 0`	`calculated`	`calculated`	`fixedValue 0`	`calculated`
p	`zeroGradient`	`slip`	`slip`	`zeroGradient`	`zeroGradient`

Here, p is pressure and U is mainstream velocity. We choose the `slip` conditions on our top wall and side walls as it ignores the normal components of the specified parameter, allowing the tangential components of the parameter to be unaffected. The `noSlip` condition on the floor is equivalent to the `fixedValue 0` boundary condition. `fixedProfile` is a logarithmic velocity profile we created for our initial velocity condition that closely resembles the Ortiz et al. paper's mean velocity figure. In Figure 4.4, we observe the comparison of our logarithmic velocity profile against Ortiz et al. velocity profile.

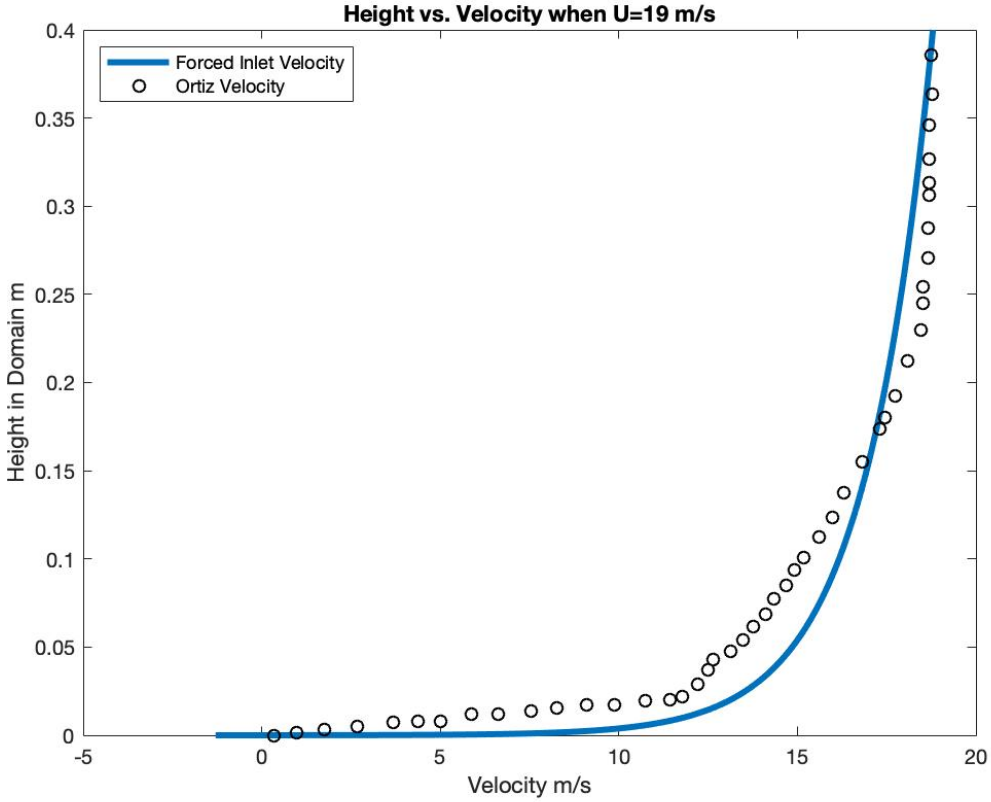

Figure 4.4: Inlet Velocity Profile for Initial Velocity Condition for Plate Simulations

4.5 Setting up Numerical Schemes and Control Dictionary

We run our simulations for 3 seconds, giving enough time for the drag and lift coefficients to converge. We choose a time step of 0.002 seconds, and we record our drag and lift coefficients at each time step. We set our velocity magnitude to be 19 m/s, our reference area to be the total surface area of the plate, 0.0264 m^2, and our reference length to be the thickness of the plate, 0.002 m. For our numerical schemes, we choose a steady-state, Gaussian linear scheme, and for the divergence scheme for velocity, we choose a bounded Gaussian linear upwind scheme, as discussed in Section 3.1.1.

4.5.1 Accounting for different angles of attack

We create 11 more geometries from our original plate of 0 degree angle of attack. Each geometry is rotated about the z-axis at degrees of 25, 35, 40, 45, 50, 55, 60, 65, 70, 80, and 90. Each plate maintains a center $\frac{3}{4}b + 10\Delta x$ meters above the floor, but only the 0 degree plate is exactly 0.15 meters from the inlet, meaning the 90 degree plate is $0.15 + \frac{1}{2}b$ meters

from the inlet. We copy the same snappyHexMeshDict, blockMeshDict, and controlDict for each angle of attack and run the 12 simulations separately.

4.6 OpenFOAM Simulation Results for Plate

We observe our simulated drag and lift coefficient results from using the SST k-ω turbulence model compared to Ortiz et al.'s experimental results using a wind tunnel, both as a function of angle of attack α, in Figure 4.5.

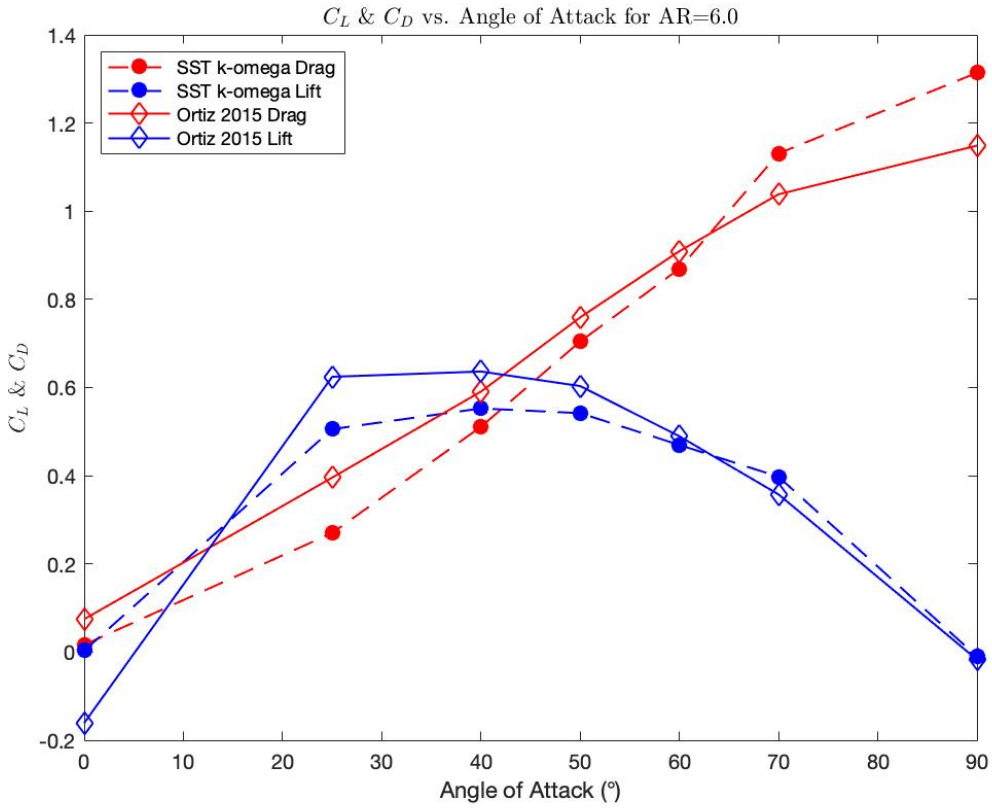

Figure 4.5: Force Coefficients for a Plate using SST k-ω vs. Ortiz's Experimental Results Dotted lines represent our results while solid lines represent Ortiz's. Blue represents lift coefficients while red represents drag coefficients.

We observe correlation between our results and Ortiz's results of approximately 0.95 for both the drag and lift coefficients. Due to this high correlation, we confirm that our settings, meshes, and our chosen turbulence model in OpenFOAM are giving accurate hydrodynamic force results.

4.6.1 Predicted versus Reported Forces Using New Drag Theory

We implement our new drag force theory discussed in Section 3.2 by first loading our calculated drag and lift coefficients from OpenFOAM into our MATLAB function `rotmatrices2DPITCH`, which uses our rotation matrices to convert our simulated forces and velocities from the computational frame of reference to the object frame of reference and then performs the regressions to calculate predictions of these forces. We observe our reported hydrodynamic forces from our OpenFOAM simulations or reported by Ortiz versus our predicted results from the MATLAB steps outlined in Section 3.2 for both our plate simulations as well as Ortiz's experimental data in Figures 4.6-7. In these figures, the vertical axes are the hydrodynamic forces in the object frame of reference, while the horizontal axes are the respective velocity components in the object frame of reference. Because this current data being examined only considers rotations around the z-axis, or pitching, we observe only the longitudinal, Ux, and normal, Uy, velocity components in the figures.

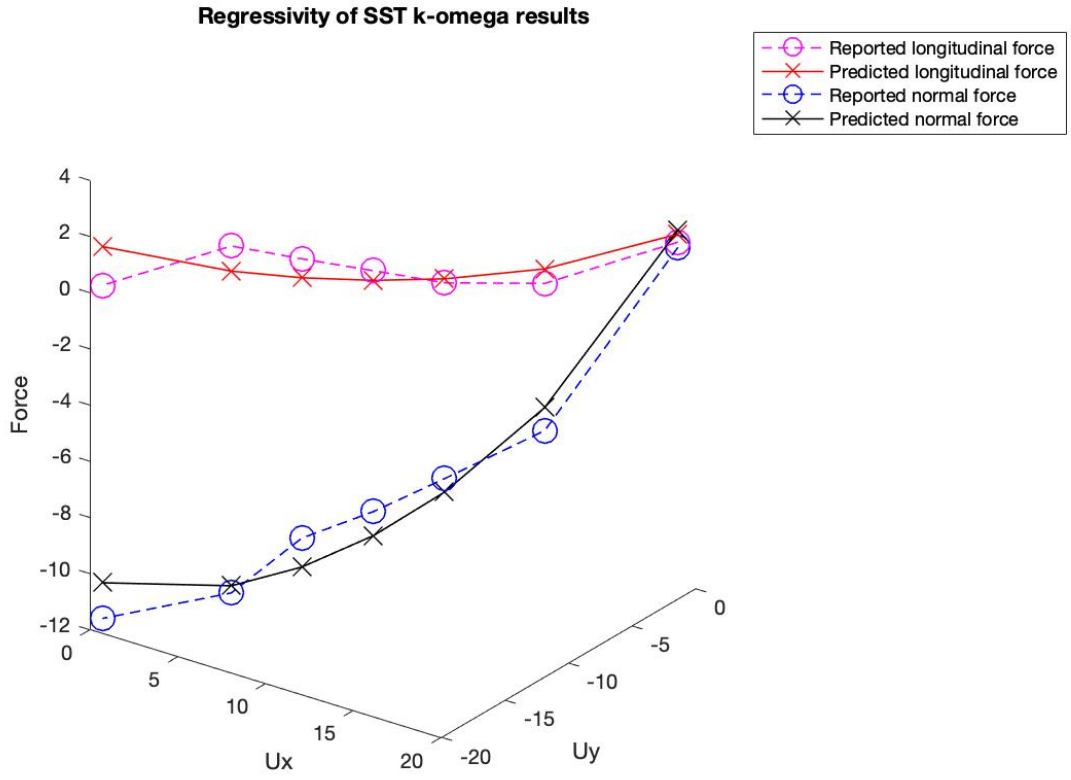

Figure 4.6: Regressivity of SST k-ω Hydrodynamic Force Results
We observe a coefficient of determination between predicted and reported forces of 0.56 for longitudinal force and 0.95 for normal force.

We report a coefficient matrix for Figure 4.6 of

$$C = \begin{bmatrix} 0.0483 & -0.1527 \\ 0.0632 & 1.1734 \end{bmatrix}.$$

To validate these coefficients, we use confidence intervals given by MATLAB's `fitlm` and `coefCI` functions. MATLAB's `fitlm` function uses QR decomposition as the main fitting algorithm, and `coefCI` uses the Wald method to solve for the confidence intervals. We observe our 95% confidence intervals of C in Table 4.3. Here, C_{Lj} denotes the longitudinal force coefficients while C_{Nj} denotes the normal force coefficients, and $j = 1$ for the force coefficient resulting from the longitudinal flow component in the object frame, u_1, and $j = 2$ for the force coefficient resulting from the normal flow component in the object frame, u_2.

Table 4.3: *95% Confidence Intervals of Figure 4.6 C.*

C	CI	Interval Size
C_{L1}	[-0.2024, 0.8858]	1.0882
C_{L2}	[-0.9806, 0.0926]	1.0732
C_{N1}	[-0.0309, 0.9854]	1.0163
C_{N2}	[0.2611, 1.2634]	1.0023

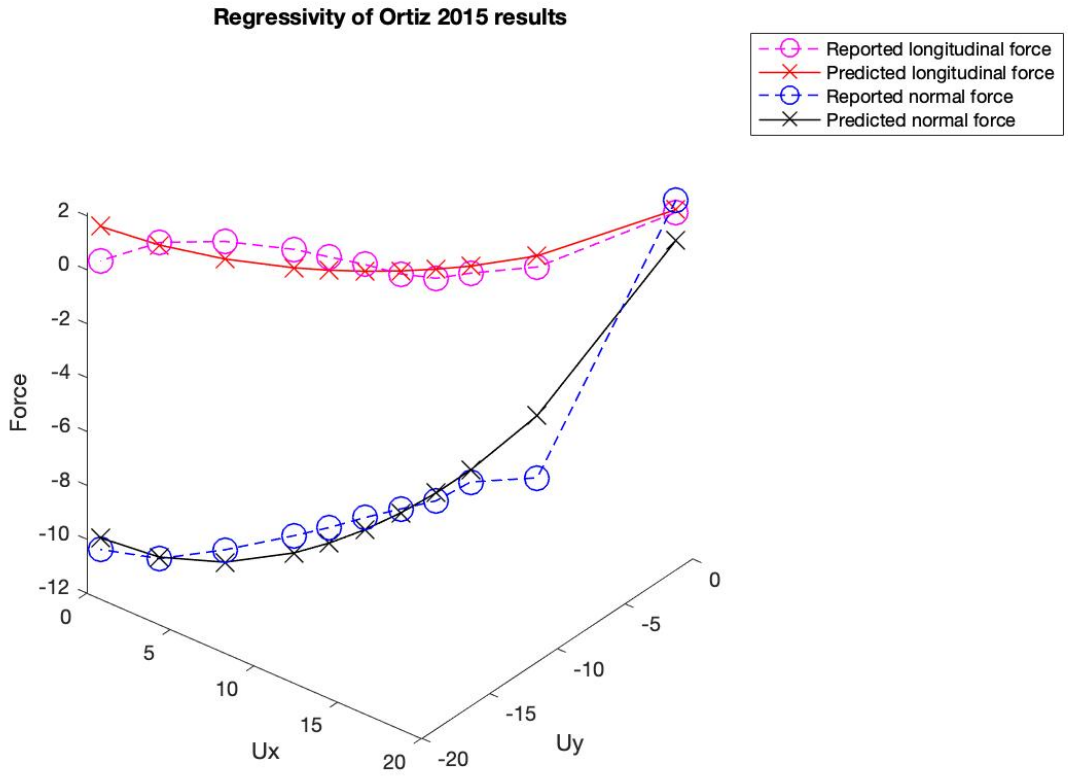

Figure 4.7: Regressivity of Ortiz's Experimental Wind Tunnel Hydrodynamic Force Results We observe a coefficient of determination between predicted and reported forces of 0.44 for longitudinal force and 0.90 for normal force.

We report a coefficient matrix for Figure 4.7 of

$$C = \begin{bmatrix} 0.0767 & -0.1411 \\ -0.0499 & 1.1394 \end{bmatrix}.$$

We observe our 95% confidence intervals of C in Table 4.4.

Table 4.4: *95% Confidence Intervals of Figure 4.7 C*.

C	**CI**	**Interval Size**
C_{L1}	[-0.0357, 0.5372]	0.5729
C_{L2}	[-0.6305, -0.0258]	0.6563
C_{N1}	[-0.7124, 0.3639]	1.0763
C_{N2}	[0.7051, 1.8411]	2.5462

We are satisfied with our coefficients of determination reported for the predicted normal forces in our regression models ($90\% - 95\%$). For our predicted longitudinal forces, however, we observe much smaller coefficients of determination ($44\% - 56\%$). Interestingly, we observe large 95% confidence intervals for our estimated normal force coefficients (> 1.0) and smaller 95% confidence intervals for our longitudinal force coefficients (< 1.0). We hypothesize that these large confidence intervals will be alleviated with more data points, and we hope to see larger coefficients of determination for our predicted longitudinal forces with this as well.

Chapter 5

OPENFOAM SIMULATIONS AND FORCE MODELS FOR LIMPET SHELLS

5.1 Introduction

Now that we have validated our use of the SST $k-\omega$ turbulence model, we wish to use OpenFOAM to calculate our drag and lift force coefficients on a limpet patella shell at 103 different orientations and various Reynolds numbers. For orientations, we will rotate our shell on each axis at degrees of 0, 45, 90, 135, and 180, resulting in 103 new models to run our simulations on. Orienting our shells will be discussed further in Section 5.2.1. For our Reynolds numbers, we would like to consider a range from 12,000 to 100,000. These Reynolds numbers correspond to flow velocities of 0.15 m/s to 1.25 m/s, respectively, for our specific limpet model and a kinematic viscosity ν of water of 10^{-6} m^2/s. In this dissertation, we only discuss results from a Reynolds number of 12,000 as more simulations for other Reynolds numbers are still running in OpenFOAM.

5.2 Obtaining and Improving a Limpet Shell Geometry

We obtain a stereolithography file of a limpet patella modeled by Emily Hauf of the Paleontological Research Institution in Ithaca, New York [8]. We scale the model to be 0.08 meters long at its longest length, therefore replicating a model close to one of the limpet shells used by Denny. Next, we must compute Δx in order to know the needed cell size to coarsen our model to, for the purpose of resolving our boundary layers with a $O(y^+ = 1)$ grid size. We follow a similar procedure to that in Section 4.2.1 to coarsen our shell model according to the Δx needed for our largest Reynolds number of 100,000, where $u = 1.25\ m/s$. Again, using a combination of the secant method and bisection similar to equation (4.2), we obtain

$$u = 2.5u_* \ln\left(9y\frac{u_*}{\nu}\right) = 2.5u_* \ln\left(9 \times 0.012\frac{u_*}{10^{-6}}\right) \longrightarrow u_* = 0.057\ m/s,$$

and

$$\Delta x = \frac{\nu}{u_*} = \frac{10^{-6}}{0.057} = 1.8 \times 10^{-5}\ m. \tag{5.1}$$

By choosing the Δx that results from our largest Reynolds number, we ensure that we are resolving the boundary layers with the finest mesh in our Reynolds number range. Therefore,

our boundary layers will still be resolved with Δx from equation (5.1) even in our simulations that use a Reynolds number of 12,000. We coarsen our limpet model according to Δx from equation (5.1), shown in Figure 5.1.

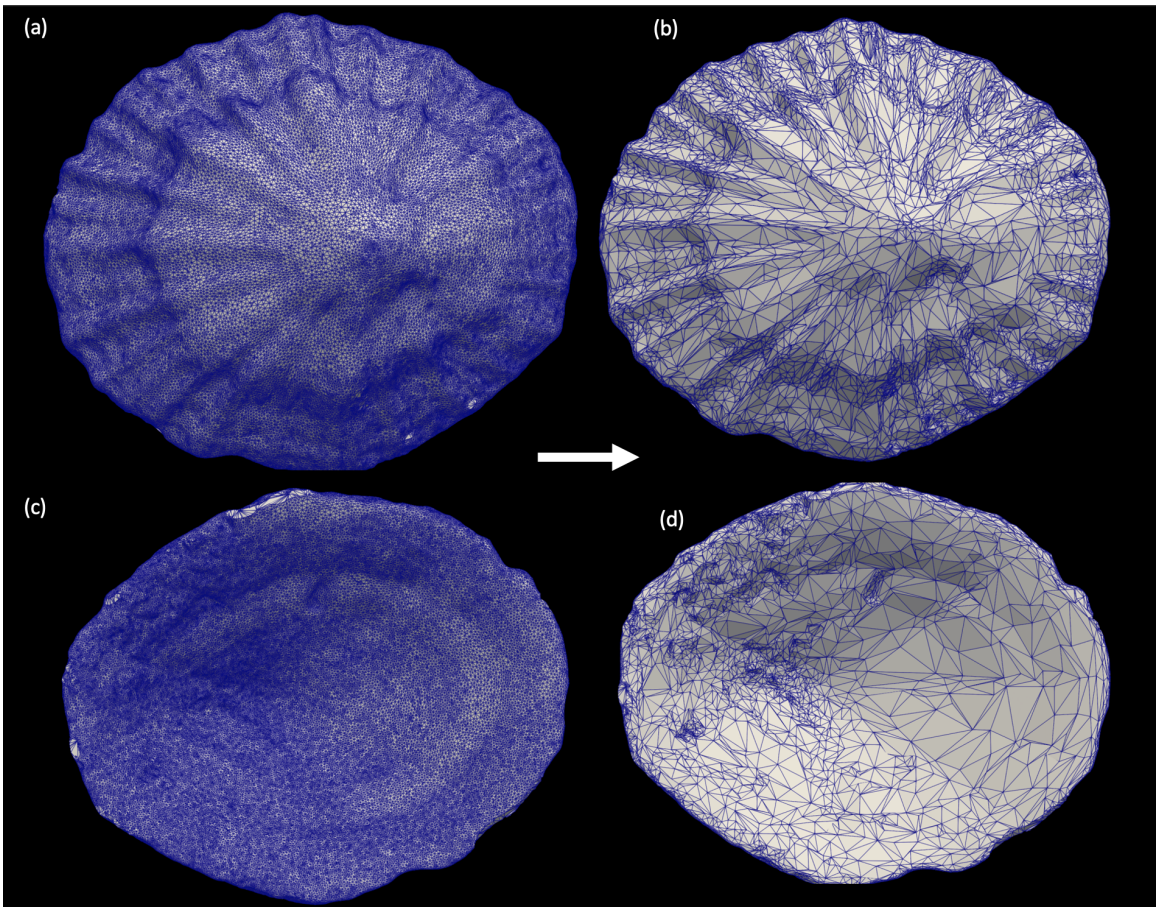

Figure 5.1: Coarsened Limpet Shell Geometry
(a) Convex side of original limpet patella shell model borrowed from the Paleontological Research Institution with a very fine mesh. **(b)** Convex side of our coarsened model with average triangle cell length of $\Delta x = 1.8 \times 10^{-5}\ m$. **(c)** Concave side of original limpet shell model. **(d)** Concave side of our coarsened model.

The surface mesh or our limpet model will now be able to resolve the boundary layers in our simulations without the use of wall functions. We will ensure that our domain mesh will also resolve the boundary layers in Section 5.3.

5.2.1 Creating Orientations of our Limpet Shell Geometry

Now that we have a shell geometry with a grid size that will resolve our boundary layers in our simulations, we must create 102 new geometries that are orientations of this initial

geometry. For organizational purposes, we assign each orientation degree a letter:

$$\begin{bmatrix} 0 & 45 & 90 & 135 & 180 \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \end{bmatrix}.$$

Then, when creating new geometries, we will name each one a combination of letters where the first letter corresponds to their x-axis orientation, the second to their y-axis orientation, and the third to their z-axis orientation. For example, if we have a shell orientation of

$$\begin{pmatrix} \mathrm{X}\ \mathrm{Y}\ \mathrm{Z} \end{pmatrix}^{\circ} = \begin{pmatrix} 0 & 90 & 135 \end{pmatrix}^{\circ},$$

it will be named **ACD**. We name our initial orientation, which is concave side down with the anterior facing the inlet, **AAA**, corresponding to an orientation of $(0\ 0\ 0)^{\circ}$.

To create the rest of our oriented geometries, we run our MATLAB function `removedups.m` that creates a shell script called `orientscript.sh`. This script is run inside a folder with the initial orientation **AAA** and creates 102 new geometries from **AAA**, each with a different permutation, choosing 3 of [0 45 90 135 180], and its corresponding name. The 103 geometries omit any duplicated geometries, such as **AAA** and **EEE**, or $(0\ 0\ 0)^{\circ}$ and $(180\ 180\ 180)^{\circ}$, respectively. The duplication is always the one removed, so there is no geometry called **EEE**.

Next, we run another script called `bbox.sh` which prints the bounding box of each new orientation into a text file called `bbox.txt`. We call another MATLAB function `translatescript.m` that reads `bbox.txt` and creates a new shell script called `translatescript.sh`. When run inside the folder with all oriented geometries, `translatescript.sh` translates each geometry to be 0.15 meters from the inlet, $10 \times \Delta x$ meters above the floor, and in the center of the spanwise direction of our domain, which will be discussed in the next section. We now have all 103 new oriented shell geometries to be used in our simulations.

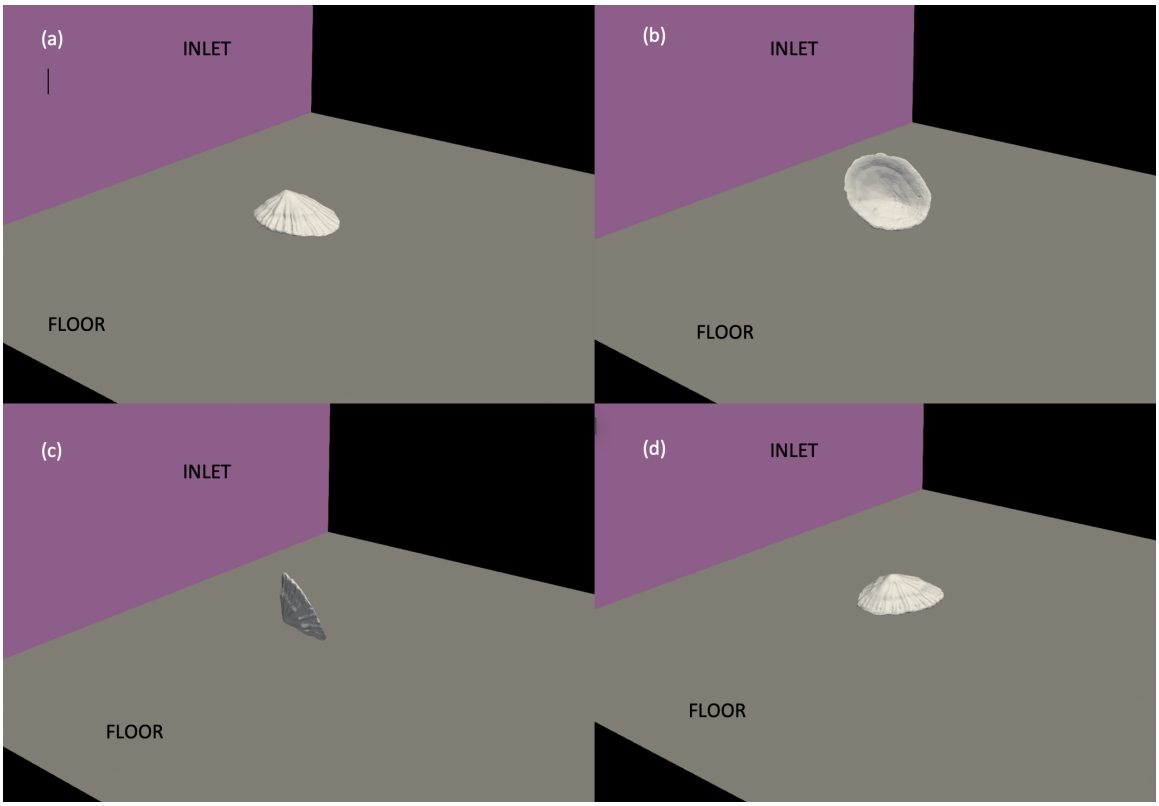

Figure 5.2: Examples of Orientations of Limpet Shell Geometry
Some examples of our oriented shell geometries sitting on our domain floor, also showing relative distance to the inlet wall. **(a)** Initial limpet geometry **AAA (b)** Shell geometry **BAD (c)** Shell geometry **CBB (d)** Shell geometry **ACA**

5.3 Setting up blockMesh and snappyHexMesh Dictionaries

We create a blockMesh domain with dimensions $0.72 \times 0.56 \times 0.40$ meters and include 6 refinement levels on the floor and wake of the shells. We use a shell script to iteratively edit the snappyHexMesh dictionary for each orientation to have a wake refinement box that is about 1.167 times the height of the respective shell geometry, or just $\frac{1}{6}$ of the shell's height above the shell, using each geometry's new bounding box dimensions. The wake refinement box is shown in Figure 5.4. We ensure the the wake refinement begins at the mid-length of each shell and extends out about 0.13 meters. The floor refinement comes to be just over the beginning height of each shell, about 0.005 meters, and of course begins at 0 meters on the y and x-axes. The floor refinement spans a width of 0.28 meters, centered in the spanwise direction, and the wake refinement spans a width of 0.12 meters, also centered in the spanwise direction.

Next, we wish to add surface layers to our surface mesh. To ensure our domain mesh is resolving the boundary layers, we need an overall surface layer thickness of $\Delta x = 1.8 \times 10^{-5}\ m$ from equation (5.1). Therefore, we set up our blockMesh to begin with cell sizes of about $1.6 \times 10^{-3}\ m$. From there, the sixth refinement level for the floor and wake will reach a smallest refined cell size of about $2.5 \times 10^{-5}\ m$. Then, in our snappyHexMesh dictionary, we set our final layer thickness to be 0.4 and our expansion ratio to be 1.2, resulting in an overall surface layer thickness on our surface mesh to be our desired $\Delta x = 1.8 \times 10^{-5}\ m$. The final layer thickness is a ratio multiplied by the cell size of the last refinement level, creating the cell size of the second surface layer. From there, the second surface layer is divided by the expansion ratio, giving the first surface layer cell size. Table 5.1 demonstrates the refinement of our cell sizes starting from our blockMesh cells, or refinement level 0, all the way down to the first surface layer. Our domain mesh will now resolve the boundary layers in our simulations without the use of wall functions.

Table 5.1: *Refinement Levels and Surface Layers in Shell Simulation Mesh.*

Refinement Levels

Level	Cell Size (m)
0	0.0015712
1	0.0007856
2	0.0003928
3	0.0001964
4	0.0000982
5	0.0000491
6	0.0000246

Surface Layers

Layer	Cell Size (m)
2	0.00000982
1	0.00000818

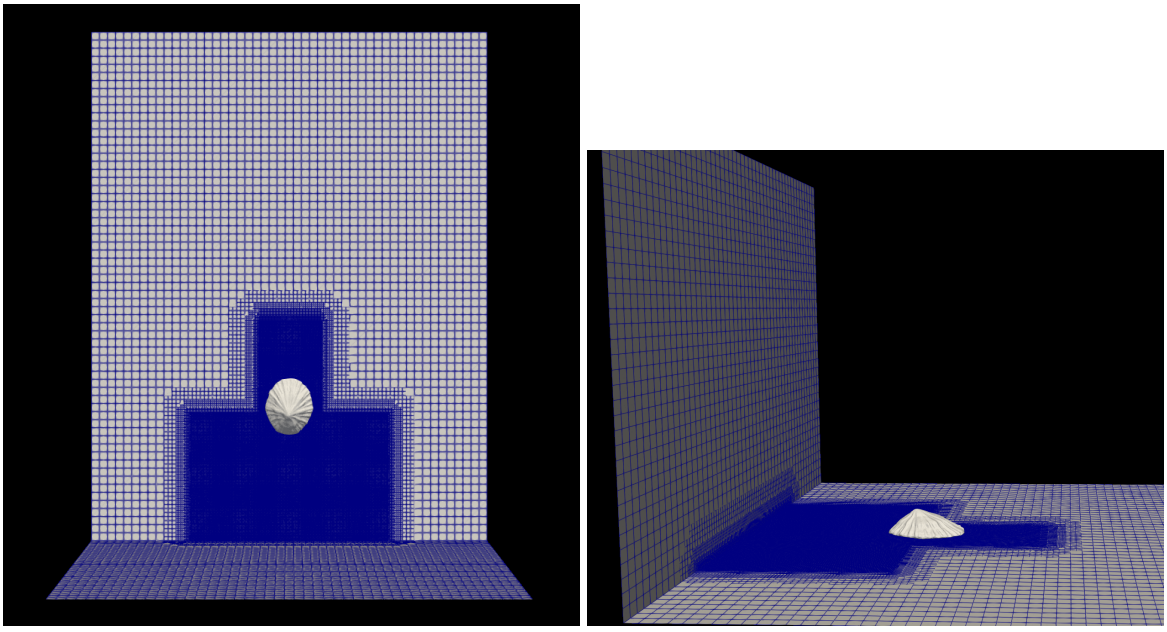

Figure 5.3: Shell Geometry **AAA** in Refined Domain
(left) Aerial view of shell **AAA** sitting on our refined floor relative to the inlet wall to the south. **(right)** Side view of shell **AAA** sitting on refined floor relative to inlet wall to the west.

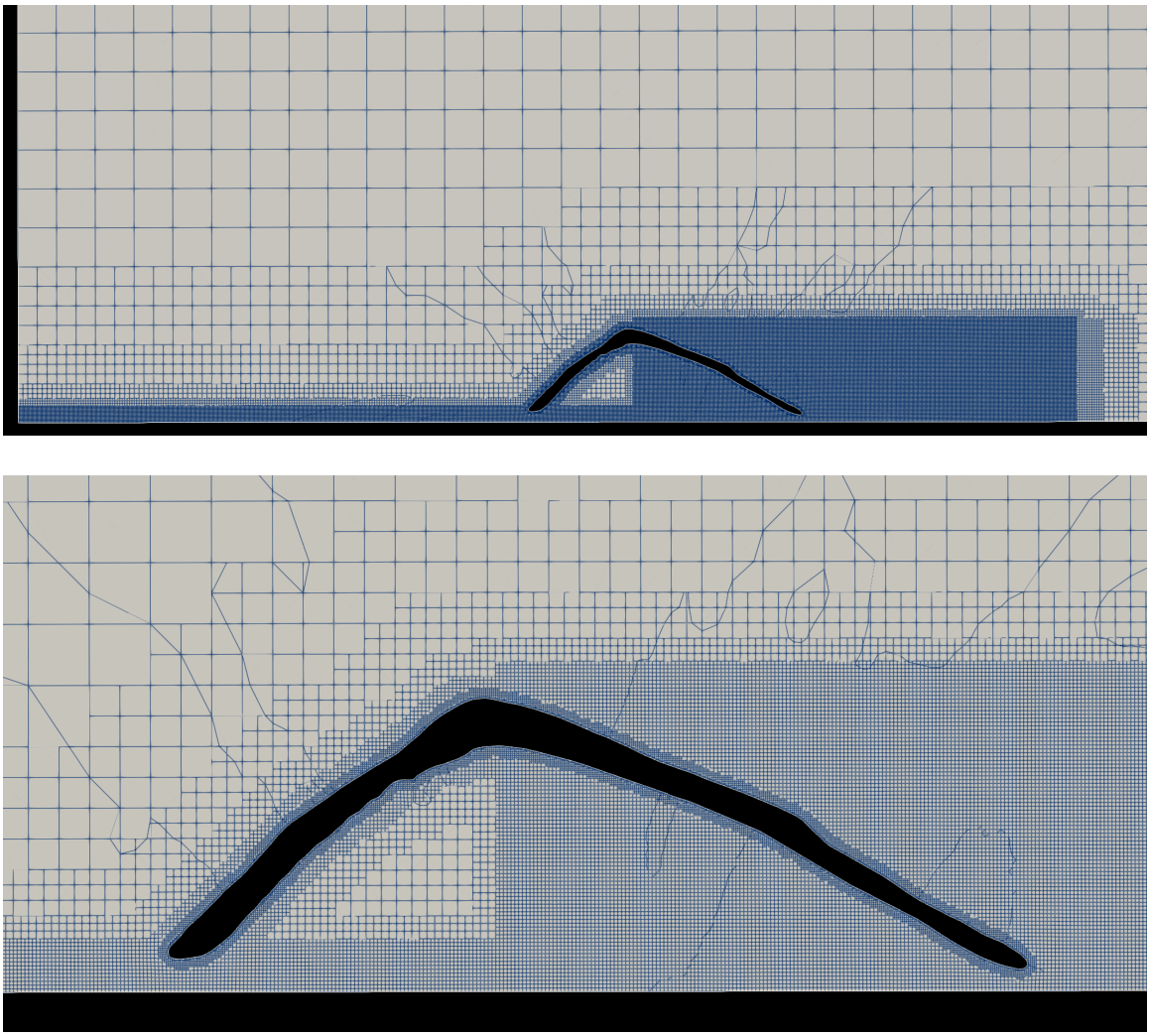

Figure 5.4: Cross-Section of Shell Geometry **AAA** in Refined Domain
Cross-section of the *xy*-axes showing shell **AAA** with the six levels of wake refinement and six levels of floor refinement.

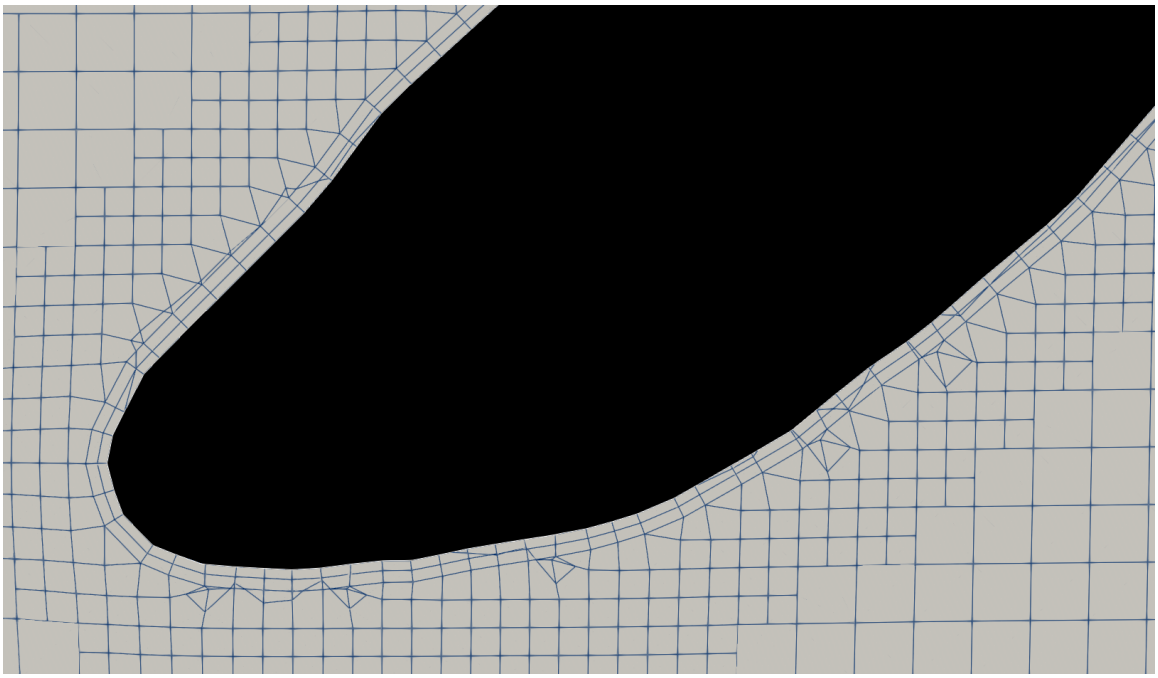

Figure 5.5: Cross-Section of Surface Layers on Shell
Cross-section of *xy*-axes showing the two surface layers on our shell's surface mesh with an overall thickness of $1.8 \times 10^{-5}\ m$.

5.4 Setting up Initial and Boundary Conditions

We calculate our initial conditions per Reynolds number using equations provided by the SST $k-\omega$ turbulence model properties, similar to Section 4.4. Here, we show the initial conditions for a Reynolds number of 24,000.

Table 5.2: *Initial Conditions for Shell Simulations with* $Re = 24,000$.

Top channel velocity U	$U = 2.5u_* ln\left(9H\frac{u_*}{\nu}\right)$	$U = 0.438\ m/s$
Turbulence length scale T	$T = 0.4H$	$T = 0.16$
Channel Reynolds number Re_C	$Re_C = \frac{HU}{\nu}$	$Re_C = 175,380$
Intensity I	$I = T(Re_C^{-0.125})$	$I = 3.54\%$
Turbulence kinetic energy k	$k = 1.5(UI)^2$	$k = 0.00036\ m^2/s^2$
Turbulence specific dissipation rate ω	$\omega = \frac{k^{0.5}}{0.55T}$	$\omega = 0.21672\ s^{-1}$

Next, we set up our boundary conditions in OpenFOAM similar to how we did for our plate simulations in Section 4. These boundary condtions remain constant for each Reynolds number and orientation of our shell simulations.

Table 5.3: *Boundary Conditions for Shell Simulations*.

Parameter	Floor	Top Wall	Side Walls	Plate	Inlet
U	`noSlip`	`slip`	`slip`	`fixedValue (0 0 0)`	`fixedProfile`
k	`fixedValue 0`	`slip`	`slip`	`fixedValue 0`	`fixedValue` from Table 5.2
ω	`fixedValue` ∞	`slip`	`slip`	`fixedValue` ∞	`fixedValue` from Table 5.2
ν_t	`fixedValue 0`	`calculated`	`calculated`	`fixedValue 0`	`calculated`
p	`zeroGradient`	`slip`	`slip`	`zeroGradient`	`zeroGradient`

For our `fixedProfile` boundary condition for velocity U, we create respective logartihmic velocity profiles for each Reynolds number. The following figure shows these profiles for our Reynolds numbers.

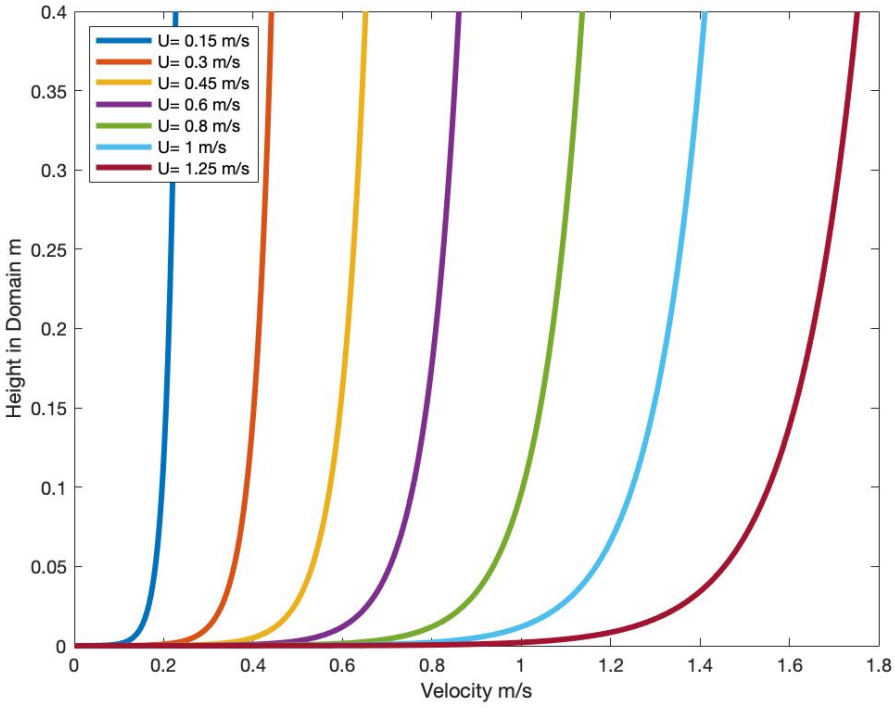

Figure 5.6: Inlet Logarithmic Velocity Profiles for Shells
Our forced inlet logarithmic velocity profiles for all seven of our considered Reynolds numbers.

5.5 Setting up Numerical Schemes and Control Dictionary

Similar to our plate simulations in Chapter 4, we choose a time step of 0.002 seconds and run each shell simulation for 3 seconds, giving enough time for convergence. We set the reference area to be the total surface area of the shells, 0.01 m^2. We choose the same numerical schemes discussed in Section 4.5.

5.6 Batching our Shell Simulations

Due to the high computational expense required for each shell simulation, we must run them in batches. In order to observe results faster, we decide to batch our simulations in groups of Reynolds numbers with all orientations. Therefore, we will begin with some results with varying orientations and a Reynolds number of 12,000.

Each simulation takes about 72 hours to run, while some can take up to 100 hours. We therefore use a script to "pack" the nodes of the HPC we are running our simulations on. This script runs as many jobs as it can on one node and keeps a queue of new jobs to add once a job finishes. Separating our batches into 4 separate jobs, we can run 10 to 11 simulations at once on one node. Reserving 4 nodes allows us to finish 42 jobs in anywhere from 72 to 100 hours.

We call a MATLAB function `nextBatch.m` that saves the drag and lift coefficient results from the last batch completed and updates each shell simulation case to use the next orientation geometry, update the snappyHexMesh to change the size of the wake refinement around the new geometry, update the initial conditions to their respective Reynolds numbers' conditions, and clean each case. Then, the next batch of 42 simulations are ready to run.

5.7 OpenFOAM Simulation Results for Shells

As discussed in Section 3.3, in order to compare our SST $k-\omega$ results for the drag and lift coefficients versus Denny's experimental results, we must scale our coefficients by a ratio of the total surface area of the shell and the surface area of the shell projected in the direction of flow. We scale our **AAA** results by a factor of 9.68 and we scale our **ACA** results by a factor of 8.75. Figure 5.7 compares our scaled results with Denny's experimental results.

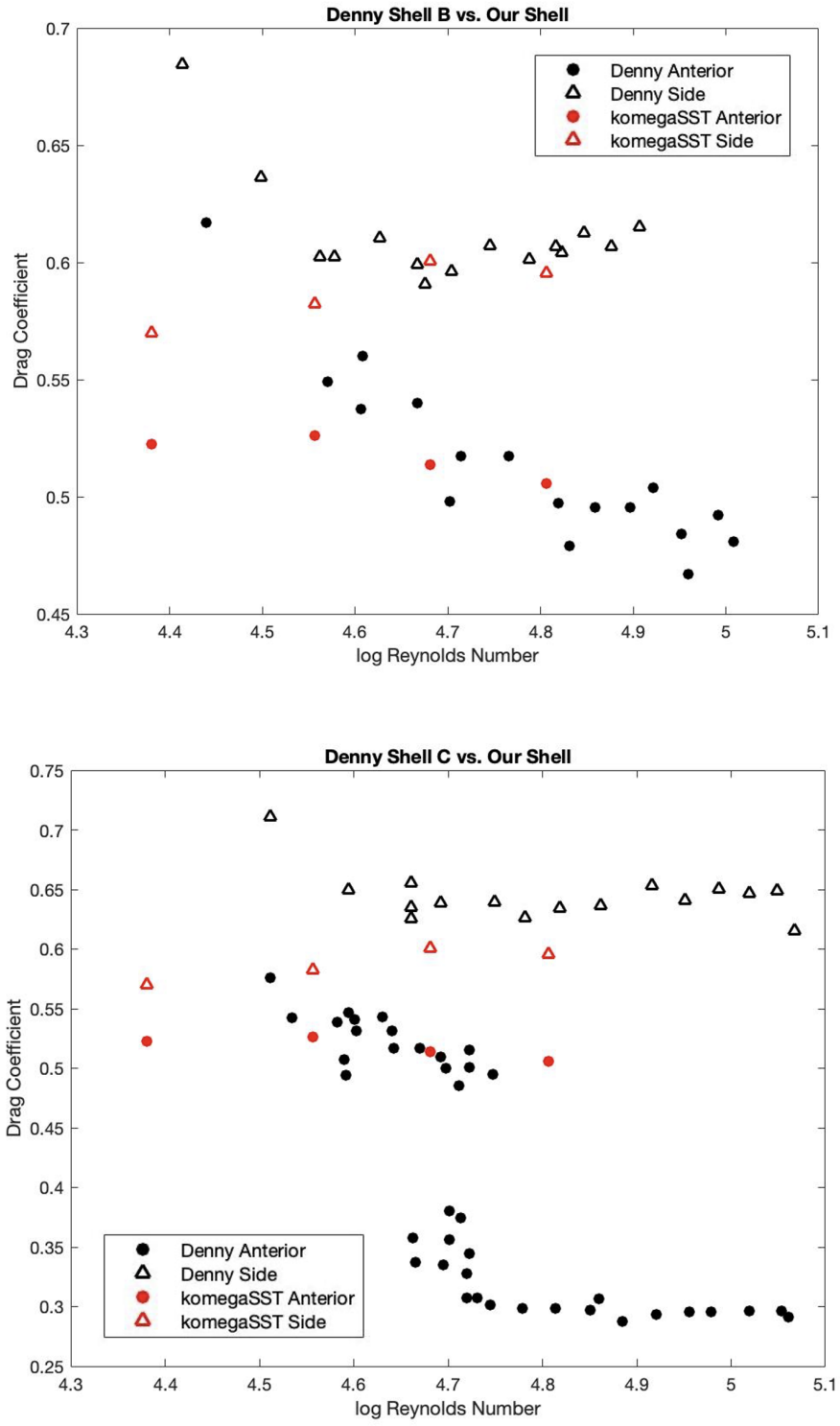

Figure 5.7: Our Scaled Drag and Lift Coefficient Results vs. Denny's Experimental Results Our scaled drag and lift coefficients for shell **AAA**, corresponding with "Anterior", and shell **ACA**, corresponding with "Side", compared against Denny's respective drag and lift coefficient results for Denny's **(top)** Shell B and **(bottom)** Shell C.

We observe a good match between our scaled, simulated drag coefficients and Denny's experimental drag coefficients. We did, however, expect to see some higher drag coefficients for our $\log(Re) = 4.4$ simulations. Overall, our drag coefficients for shell **ACA** trend upwards while our drag coefficients for shell **AAA** trend downwards, which matches with Denny's trends overall as well. For Denny's shell C, they observed a phenomenon known as the "drag crisis", or the Eiffel paradox, which occurs when the drag coefficient for a rounded object drops suddenly at large Reynolds numbers [9]. This usually occurs at a point where the flow pattern suddenly changes and a narrower turbulent wake is created behind the object. Essentially, at these higher Reynolds numbers, there is a greater separation of the boundary layer from the surface of the object. We did not observe this in our simulations, which may be due to a less rounded or smooth surface of our limpet shell compared to Denny's shell C.

Next, we observe our drag and lift coefficient results, using the total surface area for A, as a function of orientation, similar to Figure 4.5, in Figures 5.8-10.

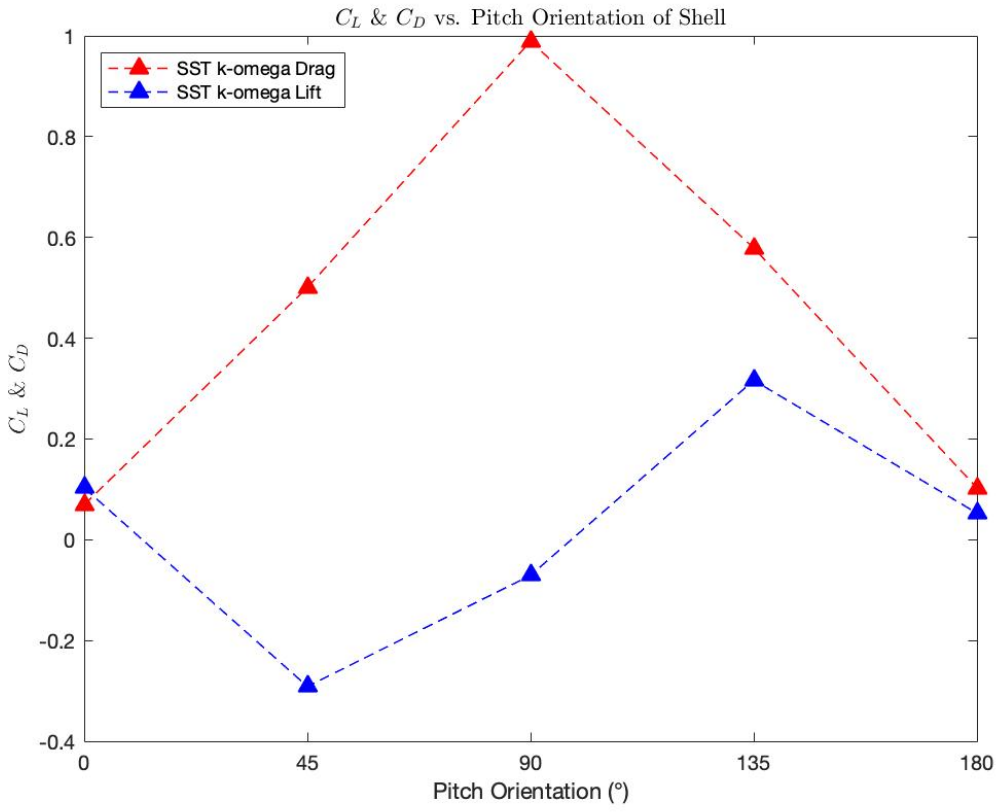

Figure 5.8: SST $k-\omega$ Drag and Lift Coefficient Results as a Function of Pitch
Our unscaled drag and lift coefficients calculated by the SST $k-\omega$ turbulence model as a function of pitching orientation.

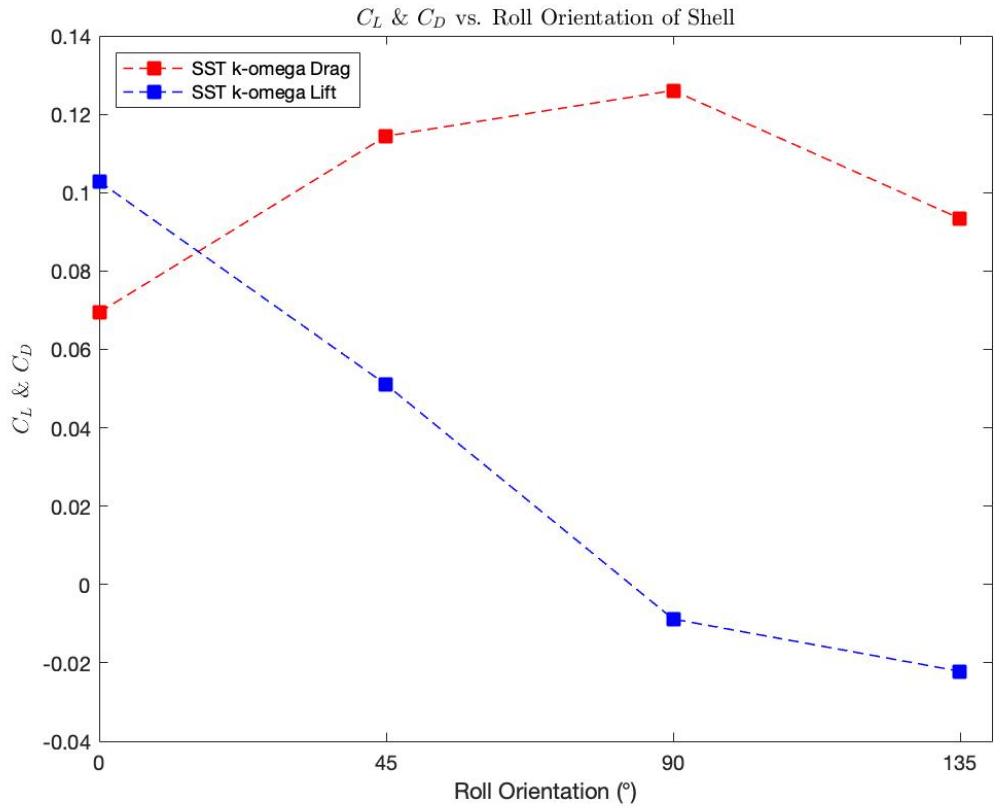

Figure 5.9: SST $k-\omega$ Drag and Lift Coefficient Results as a Function of Roll
Our unscaled drag and lift coefficients calculated by the SST $k-\omega$ turbulence model as a function of rolling orientation.

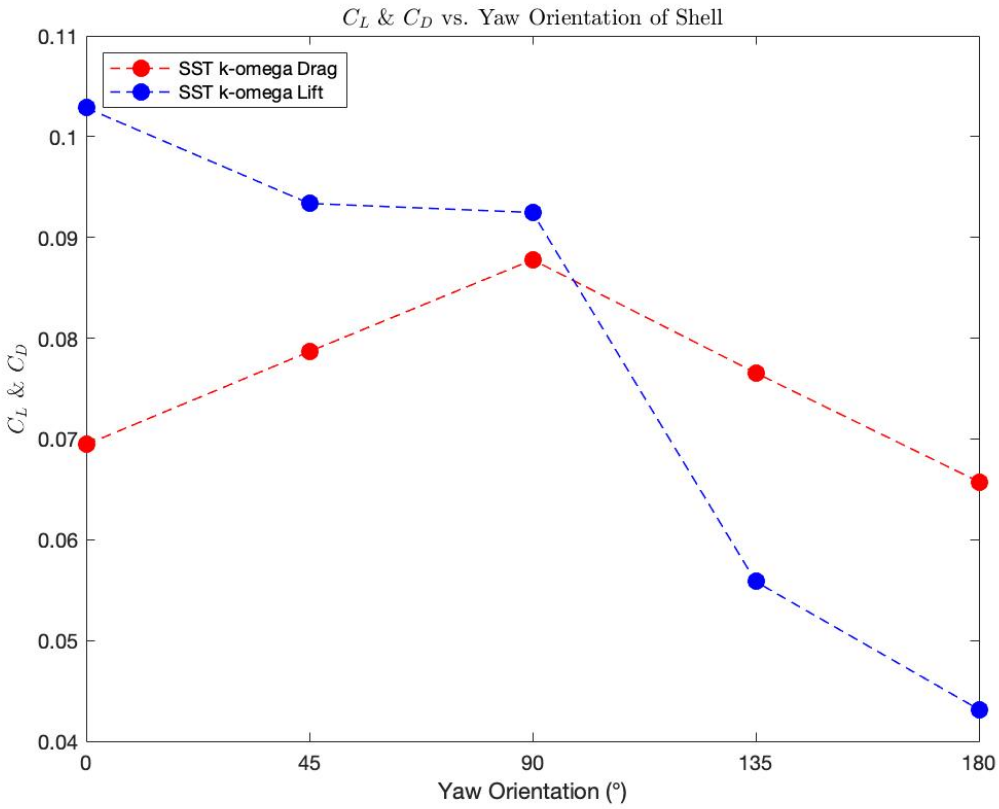

Figure 5.10: SST $k-\omega$ Drag and Lift Coefficient Results as a Function of Yaw
Our unscaled drag and lift coefficients calculated by the SST $k-\omega$ turbulence model as a function of yawing orientation.

We observe nearly parabolic trends for the drag coefficients, which was expected based off of Ortiz's results shown in Figure 4.5. Ortiz only considered angles up to 90 degrees, and they observed an upward trend of the drag coefficients as they reached this maximum angle of attack. We also observe an upward trend of the drag coefficients up to the 90 degree orientations, and then a downward trend as the shell is then oriented past 90 degrees. We notice strong symmetry in the drag coefficient trends of our yaw-oriented and pitch-oriented shells. This is expected of the yaw-oriented shell, as there is not much of a geometrical difference between the two different lateral sides of our shell. For the pitch-oriented shell, however, we wanted to observe the flow separation over time for the concave side-down shell (**AAA**) versus the convex side-down shell (**AEA**) to ensure there are no anomalies in the velocity field and the flow separations follow a similar path. We observe these results in Figure 5.11.

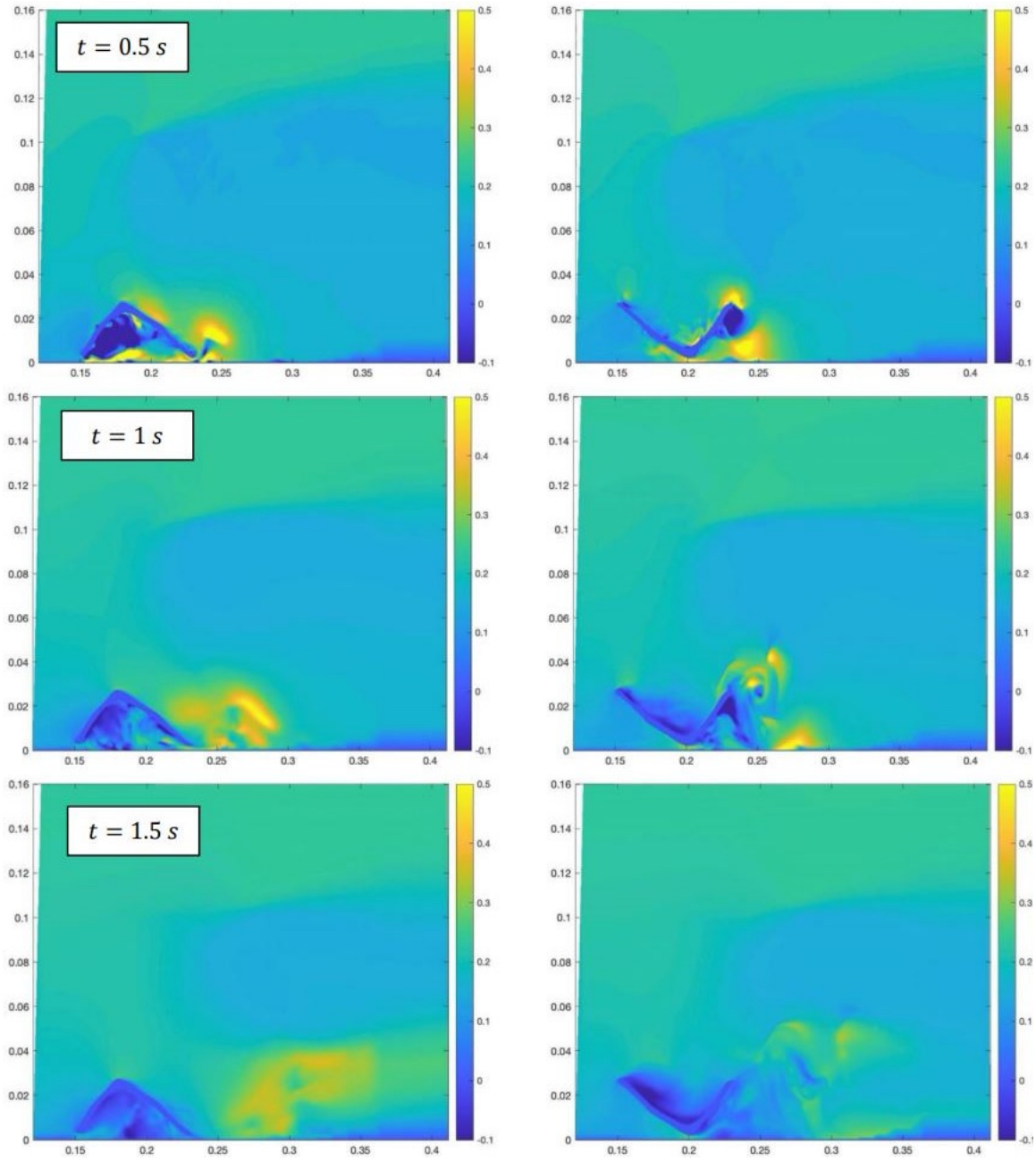

Figure 5.11: Cross-Sections of Flow Separation for **AAA** and **AEA**
Cross-sections of the *xy*-axis observing the velocity and flow separation on our shells with a pitch orientation of (**left**) 0 degrees and (**right**) 180 degrees, shown in 0.5 second time steps from the top to the bottom.

We observe no anomalies in our flow fields shown in Figure 5.11, and we notice that our flow separations, or the clouds of maximum velocities, tend to follow similar paths along the *x* and *y*-axes.

5.7.1 Predicted versus Simulated Forces Using New Drag Theory

Similar to our procedure done in Section 4.6.1, we can also observe the predicted versus simulated forces using our new drag theory discussed in Section 3.2 in Figures 5.12-14. We use MATLAB codes `rotmatrices2DPITCH`, `rotmatrices2DROLL`, and `rotmatrices2DYAW` respectively. Again, the vertical axes are the hydrodynamic forces in the object frame of reference while the horizontal axes are the respective velocity components in the object frame of reference. For varying pitch data in Figure 5.12, we only consider the longitudinal Ux and normal Uy velocity components in the object frame. For varying roll data in Figure 5.13, we only consider the spanwise Uz and normal Uy velocity components in the object frame. For varying yaw data in Figure 5.14, we only consider the spanwise Uz and longitudinal Ux velocity components in the object frame.

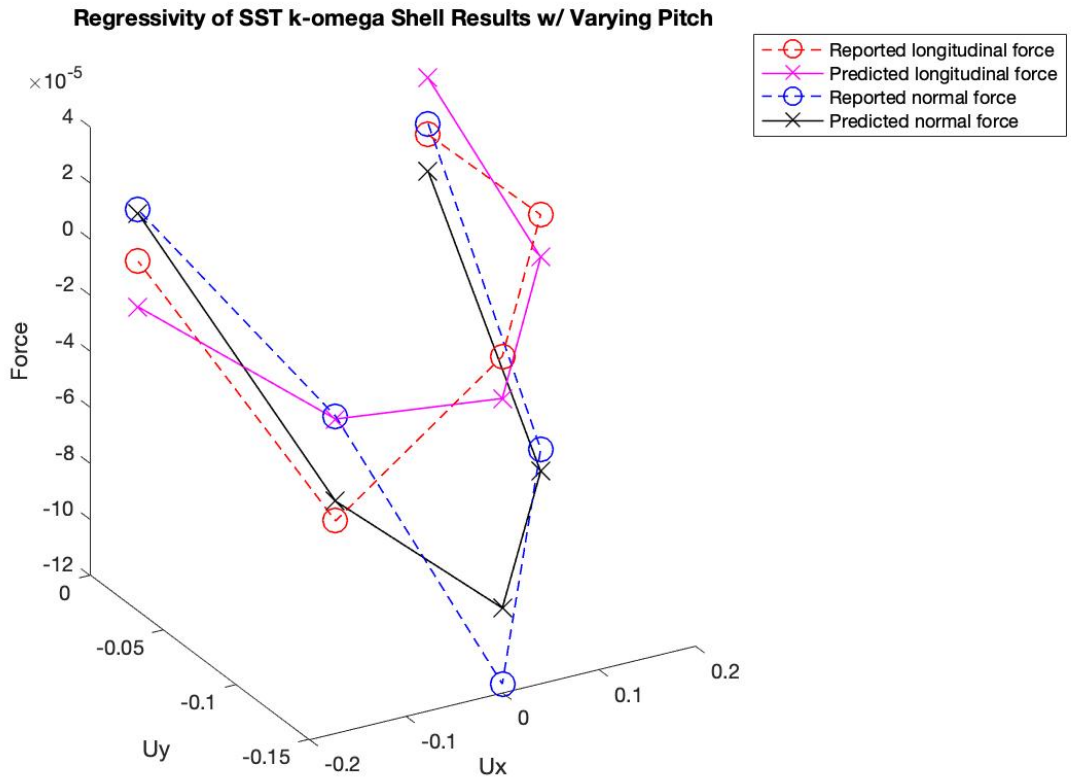

Figure 5.12: Regressivity of SST $k-\omega$ Force Results for Shells with Varying Pitch
Observation of regressivity between our predicted and simulated longitudinal and normal forces acting upon our shell in simulation with varying pitch orientations. Predicted forces result from our new drag force theory. We observe a coefficient of determination of 0.82 for normal force and 0.50 for longitudinal force.

We report a coefficient matrix for Figure 5.12 of

$$C = \begin{bmatrix} 0.2402 & 0.1270 \\ -0.0418 & 0.7600 \end{bmatrix}.$$

We observe our 95% confidence intervals of C in Table 5.4.

Table 5.4: *95% Confidence Intervals of Figure 5.12 C.*

C	**CI**	**Interval Size**
C_{L1}	[-0.4879, 0.9684]	1.4563
C_{L2}	[-1.3035, 1.4581]	2.7616
C_{N1}	[-0.6347, 0.5510]	1.1857
C_{N2}	[-0.2193, 2.0291]	2.2484

Here, C_{Lj} denotes the longitudinal force coefficients while C_{Nj} denotes the normal force coefficients.

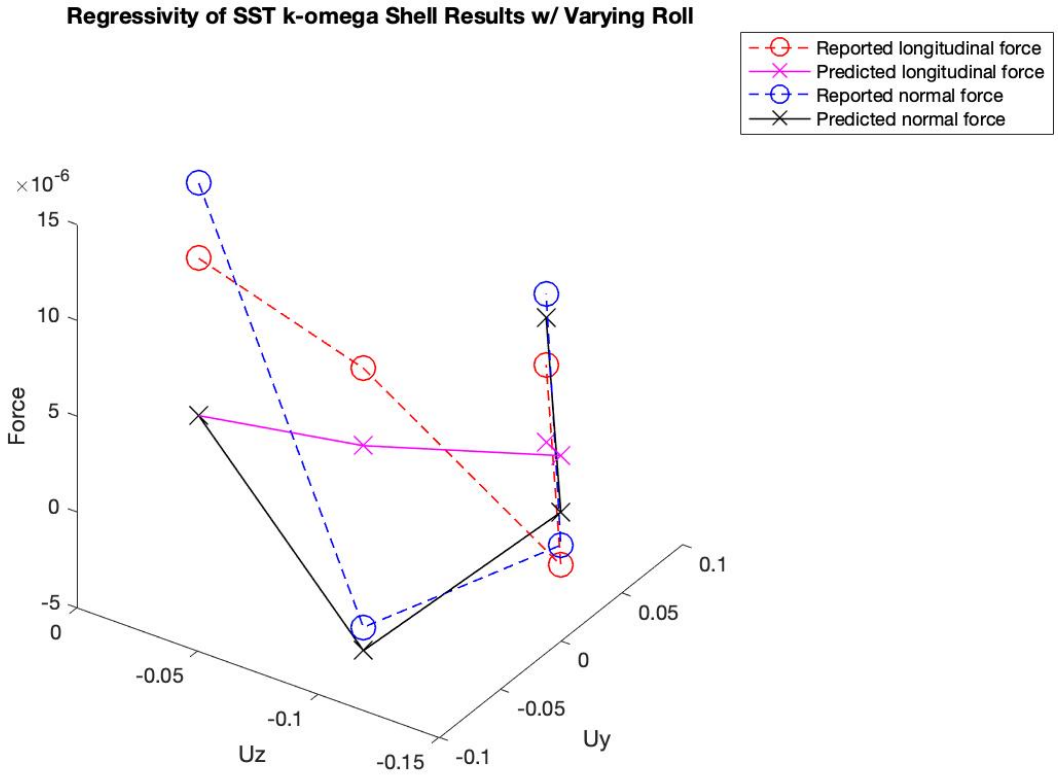

Figure 5.13: Regressivity of SST $k-\omega$ Force Results for Shells with Varying Roll Observation of regressivity between our predicted and simulated longitudinal and normal forces acting upon our shell in simulation with varying roll orientations. Predicted forces result from our new drag force theory. We observe a coefficient of determination of 0.97 for normal force and 0.65 for longitudinal force.

We report a coefficient matrix for Figure 5.13 of

$$C = \begin{bmatrix} -0.0623 & -0.0395 \\ 0.0835 & -0.0146 \end{bmatrix}.$$

We observe our 95% confidence intervals of C in Table 5.5.

Table 5.5: *95% Confidence Intervals of Figure 5.13 C.*

C	CI	Interval Size
C_{L1}	[-0.8907, 0.7662]	1.6569
C_{L2}	[-0.7351, 0.8550]	1.5901
C_{N1}	[-0.0807, 0.2477]	0.3284
C_{N2}	[-0.0511, 0.2641]	0.3152

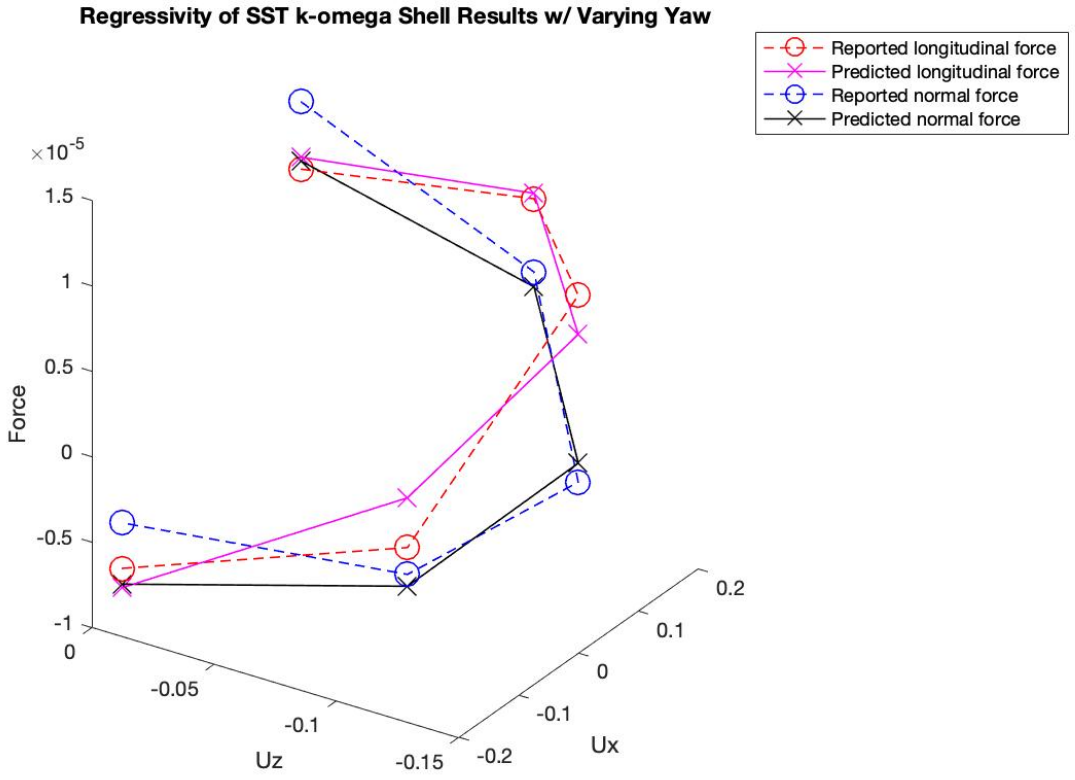

Figure 5.14: Regressivity of SST $k-\omega$ Force Results for Shells with Varying Yaw
Observation of regressivity between our predicted and simulated longitudinal and normal forces acting upon our shell in simulation with varying yaw orientations. Predicted forces result from our new drag force theory. We observe a coefficient of determination of 0.88 for normal force and 0.95 for longitudinal force.

We report a coefficient matrix for Figure 5.14 of

$$C = \begin{bmatrix} 0.0754 & -0.0756 \\ 0.0735 & 0.0274 \end{bmatrix}.$$

We observe our 95% confidence intervals of C in Table 5.6.

Table 5.6: *95% Confidence Intervals of Figure 5.14 C.*

C	CI	Interval Size
C_{L1}	[0.0169, 0.1338]	0.1169
C_{L2}	[-0.1864, 0.0352]	0.2216
C_{N1}	[0.0654, 0.0817]	0.0163
C_{N2}	[0.0120, 0.0428]	0.0308

While we are satisfied with our regression models for predicting our normal forces in all three orientation cases, we recognize low reported coefficients of determination for the longitudinal force regression models of orienting pitch and roll. We also notice large 95% confidence intervals (> 1.0) for the longitudinal force coefficients for the pitching and rolling cases. We also notice large 95% confidence intervals for the normal force coefficients of the pitching case, although we have a coefficient of determination equal to 82% of the normal force data fitting our regression model in the object frame. These results may simply be due to our relatively low number of data points in our regression models (5 for pitching and yawing and 4 for rolling).

5.7.2 Predicted versus Simulated Forces in 3D

Next, we would like to use our new force theory to perform regressions on 3-dimensional data, where the drag and lift coefficients will be represented by 3×3 matrices rather than 2×2 matrices as in previous sections. To do so, we use MATLAB code `rotmatrices3D`. We observe our predicted versus simulated force results in Figures 5.15-16 using all of the data points observed in Figures 5.12-14, which include data with varying pitch, yaw, and roll. We observe a 2D projection of these 3D regressions, where we do not include the spanwise Uz velocity component in the object frame for visualization purposes. The spanwise velocity is considered, however, when performing the regressions in the MATLAB code.

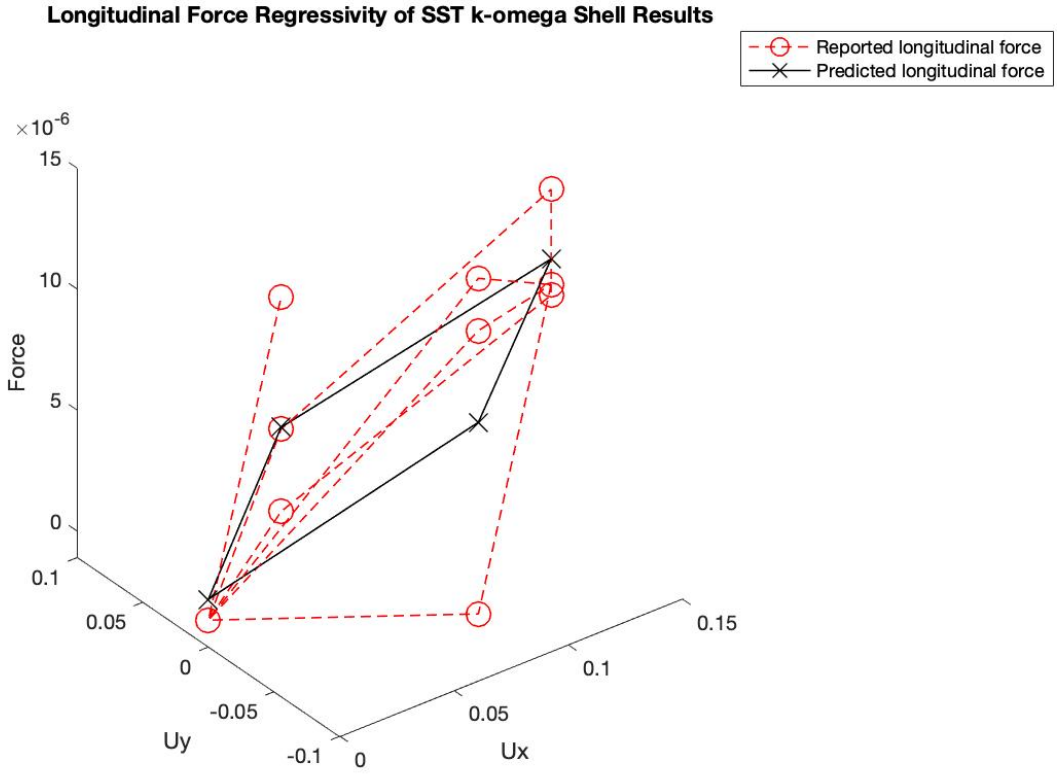

Figure 5.15: Regressivity of SST $k-\omega$ Results for Shells
Observation of the regressivity between our predicted and simulated longitudinal forces acting upon our shells with varying pitch, yaw, and roll. Predicted forces result from our new drag force theory in a 3D object frame of reference. We observe a coefficient of determination of 0.53 for longitudinal force.

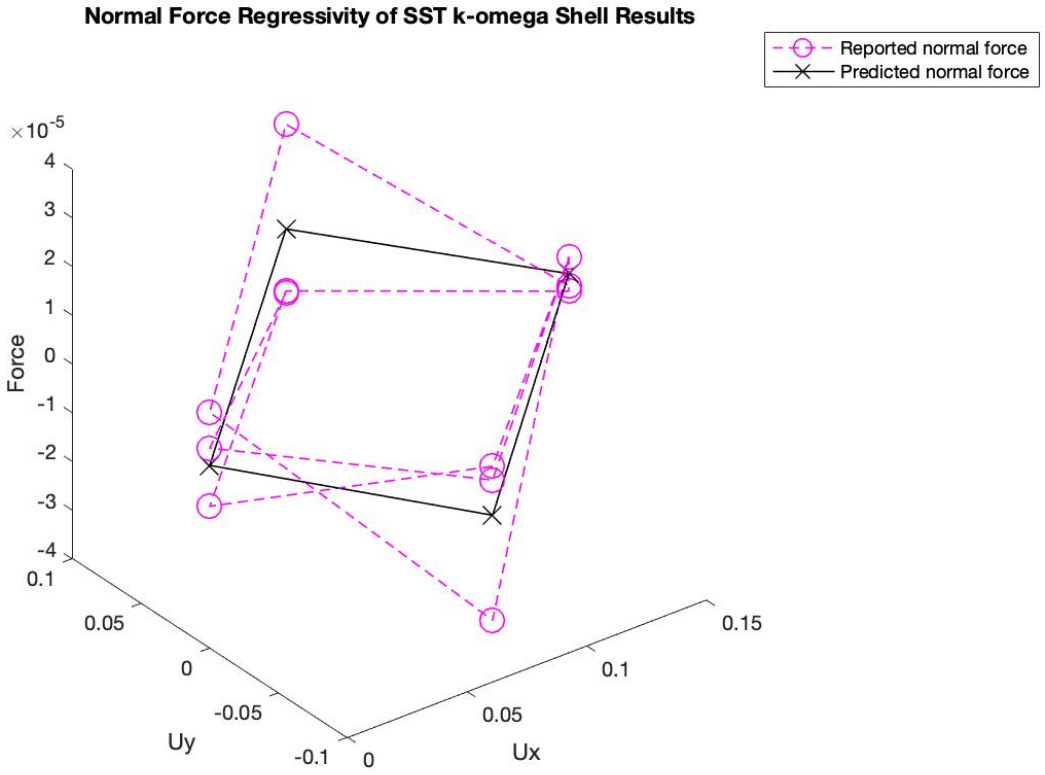

Figure 5.16: Regressivity of SST $k-\omega$ Results for Shells
Observation of the regressivity between our predicted and simulated normal forces acting upon our shells with varying pitch, yaw, and roll. Predicted forces result from our new drag force theory in a 3D object frame of reference. We observe a coefficient of determination of 0.51 for normal force.

We report a coefficient matrix for Figures 5.15 and 5.16 of

$$C = \begin{bmatrix} 0.0785 & -0.0486 & -0.0073 \\ 0.0736 & 0.2646 & 0.0214 \\ 0.0188 & -0.0469 & 0.3566 \end{bmatrix}.$$

We observe our 95% confidence intervals of C in Table 5.7.

Table 5.7: 95% Confidence Intervals of Figures 5.15-16 C.

C	CI	Interval Size
C_{L1}	[-0.0862, 0.3511]	0.4374
C_{L2}	[-0.1111, 0.0139]	0.1249
C_{L3}	[-0.2594, 0.1452]	0.4046
C_{N1}	[-0.7433, 0.6365]	1.3798
C_{N2}	[0.0676, 0.4617]	0.3941
C_{N3}	[-0.4996, 0.7769]	1.2765
C_{S1}	[-1.4895, 2.1862]	3.6757
C_{S2}	[-0.5718, 0.4781]	1.0499
C_{S3}	[-1.6477, 1.7529]	3.4006

Here, C_{Sj} denotes the spanwise force coefficients, and $j = 1$ for the spanwise force coefficient resulting from the longitudinal flow component in the object frame, u_1, $j = 2$ for the spanwise force coefficient resulting from the normal flow component in the object frame, u_2, and $j = 3$ for the spanwise force coefficient resulting from the spanwise flow component in the object frame, u_3. We observe low reported coefficients of determination for both our normal and longitudinal force regression models (51%-53%). We also observe large 95% confidence intervals (> 1.0) for our normal force coefficients and very large 95% confidence intervals for our spanwise force coefficients, some reportedly larger than 3.0. However, we observe relatively small 95% confidence intervals (< 1.0) for our longitudinal force coefficients. Again, these poor regression results may be due to a lack of data points.

Chapter 6

DISCUSSION: CONSIDERING DIFFERENT SHAPE FACTORS

6.1 The Corey Shape Factor

We wish to eventually consider a shape factor in our coefficient tensor for predicting the hydrodynamic forces on arbitrary objects. Then, our predictive model will be able to account for different limpet shells, fragments of shells, and ultimately, sand.

We will consider the Corey Shape Factor (CSF) when defining a shape factor for our objects:

$$C.S.F. = c/\sqrt{ab} \tag{6.1}$$

where c is particle thickness and a and b are particle length and width, respectively [15]. Figure 6.1 provides a illustration of these measurements on a typical shell shape.

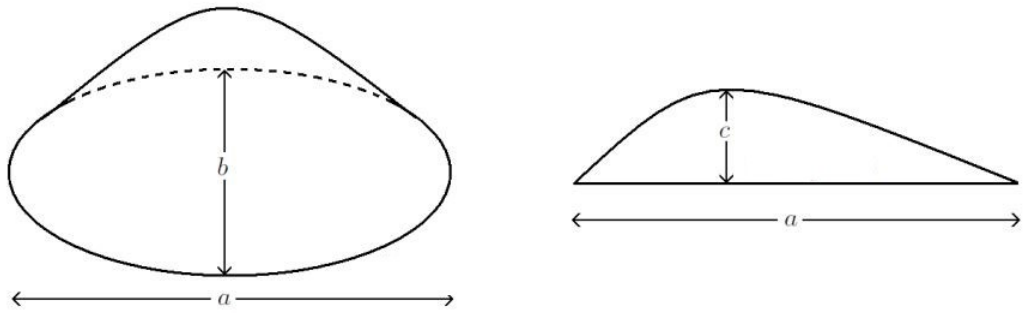

Figure 6.1: Corey Shape Factor Parameters
Illustration demonstrating the CSF parameters c, a, and b, or thickness, length, and width, respectively, on a typical shell shape.

The Zingg diagram in Figure 6.2 graphically represents different natural shapes, such as pebbles and shells, and introduces ratios $\frac{c}{b}$ as a measure of flatness and $\frac{b}{a}$ as a measure of elongation. For flatness, a value of zero represents a completely flat object. For elongation, a greater value represents a more elongated object.

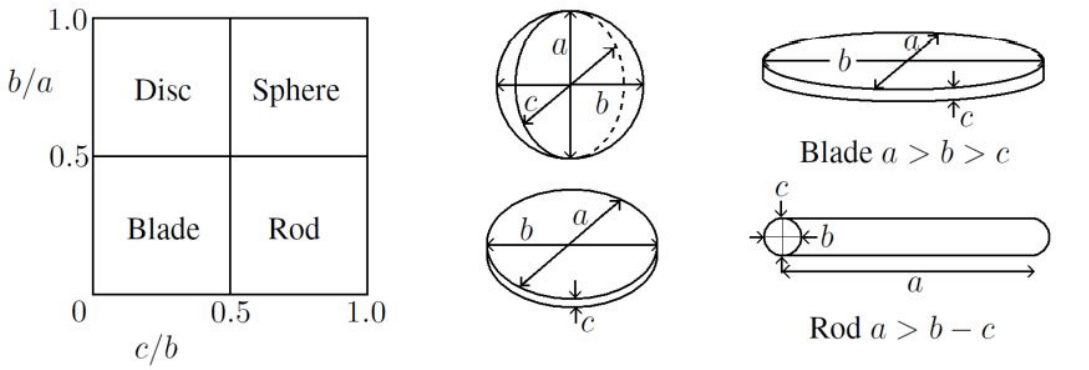

Figure 6.2: Zingg Diagram
The Zingg diagram is widely used by researchers to classify and represent shapes found in nature [15].

6.2 Using the CSF in a Tensor for Force Coefficients

After obtaining a linear regression for the forces in the object frame with various shape factors and Reynolds numbers, or

$$\hat{F}_i = \frac{1}{2} C_{i,j} \hat{u}_j \rho A ||u||,$$

we can fit the tensor coefficients as a function in the object frame of reference:

$$C_{i,j}\Big(Re, c, a, b\Big), \tag{6.2}$$

where a, b, and c are the parameters of the CSF.

6.3 Shape Factors of our Shell and Plate

We observe the shape factors of our whole limpet shell and flat plate used in our simulations, as well as some of their predicted longitudinal and normal force coefficients, in Table 6.1. For both of these shape factors, we are observing the longitudinal and normal force coefficients predicted for the varying pitch only case.

Table 6.1: *CSF, Flatness, and Elongation of our Limpet Shell and Plate*.

	CSF	Flatness	Elongation	C_{L1}	C_{N2}
Plate ($AR = 6$)	0.0394	0.0970	0.1650	0.0483	1.1734
Limpet Shell	0.3390	0.3680	0.8500	0.2402	0.7600

We observe a high normal force coefficient for the flat plate, as this would correlate with the nearly complete flatness of the plate. Our shell, which is more rounded and streamlined, reports a much lower normal force coefficient. The higher longitudinal force coefficient for the shell also is a result of the curvature of the shell.

Chapter 7
CONCLUSION

For our 2D results in Figures 4.6-7 and 5.11-13, we can conclude that our proposed force parameterization does successfully predict the normal forces with 82%-97% of the forces fitting our regression model. We report an average prediction rate of 90% and a standard deviation of 6% for the normal forces. For the longitudinal forces, we observe a lesser prediction rate with 44%-95% of the forces fitting our regression model, an average prediction rate of 62%, and a standard deviation of 20%. Our 95% confidence intervals for our coefficient matrices from these figures have an average range of 1.0480.

For our 3D results in Figures 5.14-15, we can conclude that our proposed force parameterization predicts the longitudinal and normal forces of our examined data with 53% and 51% of the forces fitting our regression model, respectively. Our 95% confidence intervals for our coefficient matrix from these figures have an average range of 1.3493.

While we knew that our parameterizations would not give us exact results, we do see potential for improvement in this method for better approximations. We recognize limitations to our force parameterization as our 95% confidence intervals seem relatively large, but implementing more orientations for each Reynolds numbers into our model should alleviate this as more data points are introduced. This is especially true for our 3D regressions, as more data points will also consider 3-dimensional rotations.

Unlike work done by Leith, we use a shape factor that works best for shapes found in nature, such as a limpet shell. We also improve parameterizations developed by Dioguardi and Denny by considering particle orientation in the computational frame of reference. We have therefore developed a new parameterization of the drag and lift coefficients on a limpet shell that depends on orientation, and more work needs to be done to implement a shape factor and more Reynolds numbers to allow this model to work for arbitrarily shaped shells and fragments. Previously, predicting the hydrodynamic forces acting on a limpet shell would have required solving an extremely complex function for the drag and lift coefficients, and we now are able to predict these forces by instead defining a tensor for the force coefficients for each unique orientation, Reynolds number, and shape factor. Due to this, we have taken a substantial step towards providing an effective parameterization to be used in Euler-Lagrange forecasting models for predicting seafloor evolution at smaller scales.

Future work will include considering more Reynolds numbers ranging from 12,000 to 100,000 as well as randomly slicing the limpet geometry into fragments and implementing the CSF into our regressions. Eventually different species of shells will be considered as well. We also intend to eventually implement this new force model into Discrete Element Method simulations that solve the Lagrangian equations of motion for each particle and carry out sediment sorting simulations.

Appendix A

Workflow

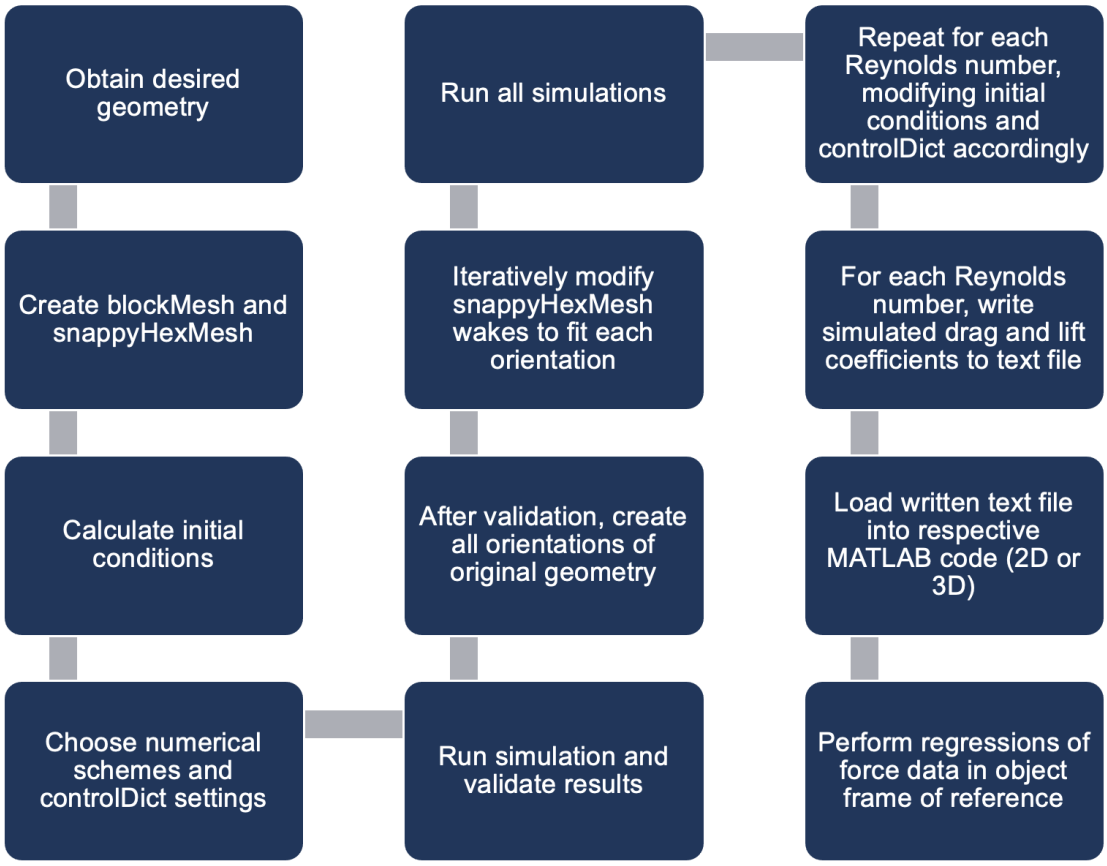

Figure A.1

Appendix B

MATLAB Codes

B.1 rotmatrices2DPITCH Code

The MATLAB code `rotmatrices2DPITCH` is used to convert the coefficient results from the computational frame of reference to the object frame of reference with only the longitudinal and normal velocity components considered. It then performs the regression that predicts the hydrodynamic forces acting on the object.

```
clear all

%load data in computational frame
Dat=load('pitchCoeffs.txt');

U=0.15; %mag velocity
A=0.0105; %area

Cd=Dat(:,1);
Cl=Dat(:,2);

R=zeros(3);
F=zeros(3,5);

AoA=[0 45 90 135 180];

for i=1:5
    alpha=AoA(i);
    beta=0;
    gamma=AoA(i);

    %%Yaw rotation matrix
    Rzi=[cosd(alpha) sind(alpha) 0; -sind(alpha) cosd(alpha) 0; 0 0 1];

    %%Pitch rotation matrix
    Ryi=[cosd(beta) 0 sind(beta); 0 1 0; -sind(beta) 0 cosd(beta)];

    %%Roll rotation matrix
    Rxi=[1 0 0; 0 cosd(gamma) -sind(gamma); 0 sind(gamma) cosd(gamma)];

    %%Unique rotation matrix
    Ri=Rzi*Ryi*Rxi;

    %Vectorize
    Rt=Ri;

    %force normalized by density
    Fd(i)=Cd(i)*U^2*A*0.5;
```

```
    Fl(i)=Cl(i)*U^2*A*0.5;

    %convert to object frame

    %rotate forces
    F(:,i)=Rt*[Fd(i) Fl(i) 0]';

    %rotate velocities
    u=U*Rt(:,1);
    Ux(i)=u(1);
    Uy(i)=u(2);
    Uz(i)=u(3);
end

%plotting longitudinal(drag) surface
figure(1)
plot3(Ux,Uy,F(1,:),'--or','LineWidth',0.8,'MarkerSize',12)

hold on
[xxd,yyd]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xd=[Ux' Uy']*A*U*0.5;
yd=F(1,:)';
dCoeffs=Xd\yd;
DCoeffs=fitlm(Xd,yd);
cid=coefCI(DCoeffs);

%predicted data
Zd=dCoeffs(1)*Xd(:,1)+dCoeffs(2)*Xd(:,2);
plot3(Ux,Uy,Zd,'-xm','LineWidth',0.8,'MarkerSize',10)

%coeff of determination
r2d=rsquare(yd,Zd);

%plotting normal(lift) surface
figure(1)
plot3(Ux,Uy,F(2,:),'--ob','LineWidth',0.8,'MarkerSize',12)
xlabel('Ux')
ylabel('Uy')
zlabel('Force')

hold on
[xxl,yyl]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xl=[Ux' Uy']*A*0.5*U;
yl=F(2,:)';
lCoeffs=Xl\yl;
LCoeffs=fitlm(Xl,yl);
cil=coefCI(LCoeffs);

%predicted data (best-fit)
Zl=lCoeffs(1)*Xl(:,1)+lCoeffs(2)*Xl(:,2);
plot3(Ux,Uy,Zl,'-xk','LineWidth',0.8,'MarkerSize',10)
legend('Reported longitudinal force','Predicted longitudinal force','Reported
normal force','Predicted normal force')
title('Regressivity of SST k-omega Shell Results w/ Varying Pitch')

%coeff of determination
r2l=rsquare(yl,Zl);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B.2 rotmatrices2DROLL Code

The MATLAB code `rotmatrices2DROLL` is used to convert the coefficient results from the computational frame of reference to the object frame of reference with only the spanwise and normal velocity components considered. It then performs the regression that predicts the hydrodynamic forces acting on the object.

```
clear all

%load data in computational frame
Dat=load('rollCoeffs.txt');

U=0.15; %mag velocity
A=0.0105; %area

Cd=Dat(:,1);
Cl=Dat(:,2);

R=zeros(3);
F=zeros(3,4);

AoA=[0 45 90 135];

for i=1:4
    alpha=AoA(i);
    beta=AoA(i);
    gamma=0;

    %%Yaw rotation matrix
    Rzi=[cosd(alpha) sind(alpha) 0; -sind(alpha) cosd(alpha) 0; 0 0 1];

    %%Pitch rotation matrix
    Ryi=[cosd(beta) 0 sind(beta); 0 1 0; -sind(beta) 0 cosd(beta)];

    %%Roll rotation matrix
    Rxi=[1 0 0; 0 cosd(gamma) -sind(gamma); 0 sind(gamma) cosd(gamma)];

    %%Unique rotation matrix
    Ri=Rzi*Ryi*Rxi;

    %Vectorize
    Rt=Ri;

    %force normalized by density
    Fd(i)=Cd(i)*U^2*A*0.5;
    Fl(i)=Cl(i)*U^2*A*0.5;

    %convert to object frame

    %rotate forces
    F(:,i)=Rt*[Fd(i) Fl(i) 0]';

    %rotate velocities
    u=U*Rt(:,1);
    Ux(i)=u(1);
```

```
    Uy(i)=u(2);
    Uz(i)=u(3);
end

%plotting longitudinal(drag) surface
figure(1)
plot3(Uy,Uz,F(1,:),'--or','LineWidth',0.8,'MarkerSize',12)

hold on
[xxd,yyd]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xd=[Uy' Uz']*A*U*0.5;
yd=F(1,:)';
dCoeffs=Xd\yd;
DCoeffs=fitlm(Xd,yd);
cid=coefCI(DCoeffs);

%predicted data
Zd=dCoeffs(1)*Xd(:,1)+dCoeffs(2)*Xd(:,2);
plot3(Uy,Uz,Zd,'-xm','LineWidth',0.8,'MarkerSize',10)

%coeff of determination
r2d=rsquare(yd,Zd);

%plotting normal(lift) surface
figure(1)
plot3(Uy,Uz,F(2,:),'--ob','LineWidth',0.8,'MarkerSize',12)
xlabel('Uy')
ylabel('Uz')
zlabel('Force')

hold on
[xxl,yyl]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xl=[Uy' Uz']*A*0.5*U;
yl=F(2,:)';
lCoeffs=Xl\yl;
LCoeffs=fitlm(Xl,yl);
cil=coefCI(LCoeffs);

%predicted data (best-fit)
Zl=lCoeffs(1)*Xl(:,1)+lCoeffs(2)*Xl(:,2);
plot3(Uy,Uz,Zl,'-xk','LineWidth',0.8,'MarkerSize',10)
legend('Reported longitudinal force','Predicted longitudinal force','Reported
normal force','Predicted normal force')
title('Regressivity of SST k-omega Shell Results w/ Varying Roll')

%coeff of determination
r2l=rsquare(yl,Zl);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B.3 rotmatrices2DYAW Code

The MATLAB code `rotmatrices2DYAW` is used to convert the coefficient results from the computational frame of reference to the object frame of reference with only the spanwise and longitudinal velocity components considered. It then performs the regression that predicts the hydrodynamic forces acting on the object.

```
clear all

%load data in computational frame
Dat=load('yawCoeffs.txt');

U=0.15; %mag velocity
A=0.0105; %area

Cd=Dat(:,1);
Cl=Dat(:,2);

R=zeros(3);
F=zeros(3,5);

AoA=[0 45 90 135 180];

for i=1:5
    alpha=0;
    beta=AoA(i);
    gamma=AoA(i);

    %%Yaw rotation matrix
    Rzi=[cosd(alpha) sind(alpha) 0; -sind(alpha) cosd(alpha) 0; 0 0 1];

    %%Pitch rotation matrix
    Ryi=[cosd(beta) 0 sind(beta); 0 1 0; -sind(beta) 0 cosd(beta)];

    %%Roll rotation matrix
    Rxi=[1 0 0; 0 cosd(gamma) -sind(gamma); 0 sind(gamma) cosd(gamma)];

    %%Unique rotation matrix
    Ri=Rzi*Ryi*Rxi;

    %Vectorize
    Rt=Ri;

    %force normalized by density
    Fd(i)=Cd(i)*U^2*A*0.5;
    Fl(i)=Cl(i)*U^2*A*0.5;

    %convert to object frame

    %rotate forces
    F(:,i)=Rt*[Fd(i) Fl(i) 0]';

    %rotate velocities
    u=U*Rt(:,1);
    Ux(i)=u(1);
```

```
    Uy(i)=u(2);
    Uz(i)=u(3);
end

%plotting longitudinal(drag) surface
figure(1)
plot3(Ux,Uz,F(1,:),'--or','LineWidth',0.8,'MarkerSize',12)

hold on
[xxd,yyd]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xd=[Ux' Uz']*A*U*0.5;
yd=F(1,:)';
dCoeffs=Xd\yd;
DCoeffs=fitlm(Xd,yd);
cid=coefCI(DCoeffs);

%predicted data
Zd=dCoeffs(1)*Xd(:,1)+dCoeffs(2)*Xd(:,2);
plot3(Ux,Uz,Zd,'-xm','LineWidth',0.8,'MarkerSize',10)

%coeff of determination
r2d=rsquare(yd,Zd);

%plotting normal(lift) surface
figure(1)
plot3(Ux,Uz,F(2,:),'--ob','LineWidth',0.8,'MarkerSize',12)
xlabel('Ux')
ylabel('Uz')
zlabel('Force')

hold on
[xxl,yyl]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xl=[Ux' Uz']*A*0.5*U;
yl=F(2,:)';
lCoeffs=Xl\yl;
LCoeffs=fitlm(Xl,yl);
cil=coefCI(LCoeffs);

%predicted data (best-fit)
Zl=lCoeffs(1)*Xl(:,1)+lCoeffs(2)*Xl(:,2);
plot3(Ux,Uz,Zl,'-xk','LineWidth',0.8,'MarkerSize',10)
legend('Reported longitudinal force','Predicted longitudinal force','Reported
normal force','Predicted normal force')
title('Regressivity of SST k-omega Shell Results w/ Varying Yaw')

%coeff of determination
r2l=rsquare(yl,Zl);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B.4 rotmatrices3D Code

The MATLAB code `rotmatrices3D` is used to convert the coefficient results from the computational frame of reference to the object frame of reference with all three velocity components considered. It then performs the regression that predicts the hydrodynamic forces acting on the object.

```
clear all

%load data in computational frame
Dat=load('allData.txt');

U=0.15; %mag velocity
A=0.0105; %area

Cd=Dat(:,1);
Cl=Dat(:,2);

R=zeros(3);
F=zeros(3,length(Dat));

alphas=[0 45 90 135 180 0 45 90 135 180 0 45 90 135];
betas=[0 45 90 135 180 0 45 90 135 180 0 45 90 135];
gammas=[0 45 90 135 180 0 45 90 135 180 0 45 90 135];
% alphas=[0 45 90 135 180];
% betas=[0 45 90 135 180];
% gammas=[0 45 90 135 180];

for i=1:length(Dat)
    alpha=alphas(i);
    beta=betas(i);
    gamma=gammas(i);

    %%Yaw rotation matrix
    Rzi=[cosd(alpha) sind(alpha) 0; -sind(alpha) cosd(alpha) 0; 0 0 1];

    %%Pitch rotation matrix
    Ryi=[cosd(beta) 0 sind(beta); 0 1 0; -sind(beta) 0 cosd(beta)];

    %%Roll rotation matrix
    Rxi=[1 0 0; 0 cosd(gamma) -sind(gamma); 0 sind(gamma) cosd(gamma)];

    %%Unique rotation matrix
    Ri=Rzi*Ryi*Rxi;

    %Vectorize
    Rt=Ri;

    %force normalized by density
    Fd(i)=Cd(i)*U^2*A*0.5;
    Fl(i)=Cl(i)*U^2*A*0.5;

    %convert to object frame

    %rotate forces
```

```
    F(:,i)=Rt*[Fd(i) Fl(i) 0]';

    %rotate velocities
    u=U*Rt(:,1);
    Ux(i)=u(1);
    Uy(i)=u(2);
    Uz(i)=u(3);

end

%plotting longitudinal(drag) surface
figure(1)
plot3(Ux,Uy,F(1,:),'--or','LineWidth',0.8,'MarkerSize',12)
xlabel('Ux')
ylabel('Uy')
zlabel('Force')

hold on
[xxd,yyd]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xd=[Ux' Uy' Uz']*A*U*0.5;
yd=F(1,:)';
dCoeffs=Xd\yd;
DCoeffs=fitlm(Xd,yd);
cid=coefCI(DCoeffs);

%predicted data
Zd=dCoeffs(1)*Xd(:,1)+dCoeffs(2)*Xd(:,2)+dCoeffs(3)*Xd(:,3);
plot3(Ux,Uy,Zd,'-xk','LineWidth',0.8,'MarkerSize',10)
legend('Reported longitudinal force','Predicted longitudinal force')
title('Longitudinal Force Regressivity of SST k-omega Shell Results')

%coeff of determination
r2d=rsquare(yd,Zd);

%plotting normal(lift) surface
figure(2)
plot3(Ux,Uy,F(2,:),'--om','LineWidth',0.8,'MarkerSize',12)
xlabel('Ux')
ylabel('Uy')
zlabel('Force')

hold on
[xxl,yyl]=meshgrid(1:20:400,1:20:400); % Generating a regular grid for plotting
Xl=[Ux' Uy' Uz']*A*0.5*U;
yl=F(2,:)';
lCoeffs=Xl\yl;
LCoeffs=fitlm(Xl,yl);
cil=coefCI(LCoeffs);

%predicted data (best-fit)
Zl=lCoeffs(1)*Xl(:,1)+lCoeffs(2)*Xl(:,2)+lCoeffs(3)*Xl(:,3);
plot3(Ux,Uy,Zl,'-xk','LineWidth',0.8,'MarkerSize',10)
legend('Reported normal force','Predicted normal force')
title('Normal Force Regressivity of SST k-omega Shell Results')

%coeff of determination
r2l=rsquare(yl,Zl);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%third force and velocity comp
Xs=[Ux' Uy' Uz']*A*0.5*U;
ys=F(3,:)';
sCoeffs=Xs\ys;
SCoeffs=fitlm(Xs,ys);
cis=coefCI(SCoeffs);

cidSize(1)=cid(2,2)-cid(2,1);
cidSize(2)=cid(3,2)-cid(3,1);
cidSize(3)=cid(4,2)-cid(4,1);

cilSize(1)=cil(2,2)-cil(2,1);
cilSize(2)=cil(3,2)-cil(3,1);
cilSize(3)=cil(4,2)-cil(4,1);

cisSize(1)=cis(2,2)-cis(2,1);
cisSize(2)=cis(3,2)-cis(3,1);
cisSize(3)=cis(4,2)-cis(4,1);
```

B.5 permn Code

The MATLAB code `permn` is used to create permuations of our orientation degrees to be used in `removedups2`.

```
function [M, I] = permn(V, N, K)
% PERMN - permutations with repetition
%   Using two input variables V and N, M = PERMN(V,N) returns all
%   permutations of N elements taken from the vector V, with repetitions.
%   V can be any type of array (numbers, cells etc.) and M will be of the
%   same type as V.  If V is empty or N is 0, M will be empty.  M has the
%   size numel(V).^N-by-N.
%
%   When only a subset of these permutations is needed, you can call PERMN
%   with 3 input variables: M = PERMN(V,N,K) returns only the K-ths
%   permutations.  The output is the same as M = PERMN(V,N) ; M = M(K,:),
%   but it avoids memory issues that may occur when there are too many
%   combinations.  This is particulary useful when you only need a few
%   permutations at a given time. If V or K is empty, or N is zero, M will
%   be empty. M has the size numel(K)-by-N.
%
%   [M, I] = PERMN(...) also returns an index matrix I so that M = V(I).
%
%   Examples:
%     M = permn([1 2 3],2) % returns the 9-by-2 matrix:
%              1     1
%              1     2
%              1     3
%              2     1
%              2     2
%              2     3
%              3     1
%              3     2
%              3     3
%
%     M = permn([99 7],4) % returns the 16-by-4 matrix:
%              99     99    99    99
%              99     99    99     7
%              99     99     7    99
%              99     99     7     7
%              ...
%               7      7     7    99
%               7      7     7     7
%
%     M = permn({'hello!' 1:3},2) % returns the 4-by-2 cell array
%             'hello!'        'hello!'
%             'hello!'        [1x3 double]
%             [1x3 double]    'hello!'
%             [1x3 double]    [1x3 double]
%
%     V = 11:15, N = 3, K = [2 124 21 99]
%     M = permn(V, N, K) % returns the 4-by-3 matrix:
%     %        11  11  12
%     %        15  15  14
%     %        11  15  11
%     %        14  15  14
```

```
%     % which are the 2nd, 124th, 21st and 99th permutations
%     % Check with PERMN using two inputs
%     M2 = permn(V,N) ; isequal(M2(K,:),M)
%     % Note that M2 is a 125-by-3 matrix
%
%     % PERMN can be used generate a binary table, as in
%     B = permn([0 1],5)
%
%   NB Matrix sizes increases exponentially at rate (n^N)*N.
%
%   See also PERMS, NCHOOSEK
%            ALLCOMB, PERMPOS, NEXTPERM, NCHOOSE2 on the File Exchange

% tested in Matlab 2018a
% version 6.2 (jan 2019)
% (c) Jos van der Geest
% Matlab File Exchange Author ID: 10584
% email: samelinoa@gmail.com

% History
% 1.1 updated help text
% 2.0 new faster algorithm
% 3.0 (aug 2006) implemented very fast algorithm
% 3.1 (may 2007) Improved algorithm Roger Stafford pointed out that for some values
%, the floor
%   operation on floating points, according to the IEEE 754 standard, could return
%   erroneous values. His excellent solution was to add (1/2) to the values
%   of A.
% 3.2 (may 2007) changed help and error messages slightly
% 4.0 (may 2008) again a faster implementation, based on ALLCOMB, suggested on the
%   newsgroup comp.soft-sys.matlab on May 7th 2008 by "Helper". It was
%   pointed out that COMBN(V,N) equals ALLCOMB(V,V,V...) (V repeated N
%   times), ALLCMOB being faster. Actually version 4 is an improvement
%   over version 1 ...
% 4.1 (jan 2010) removed call to FLIPLR, using refered indexing N:-1:1
%   (is faster, suggestion of Jan Simon, jan 2010), removed REPMAT, and
%   let NDGRID handle this
% 4.2 (apr 2011) corrrectly return a column vector for N = 1 (error pointed
%    out by Wilson).
% 4.3 (apr 2013) make a reference to COMBNSUB
% 5.0 (may 2015) NAME CHANGED (COMBN -> PERMN) and updated description,
%   following comment by Stephen Obeldick that this function is misnamed
%   as it produces permutations with repetitions rather then combinations.
% 5.1 (may 2015) always calculate M via indices
% 6.0 (may 2015) merged the functionaly of permnsub (aka combnsub) and this
%   function
% 6.1 (may 2016) fixed spelling errors
% 6.2 (jan 2019) fixed some coding style warnings

narginchk(2, 3) ;

if fix(N) ~= N || N < 0 || numel(N) ~= 1
    error('permn:negativeN','Second argument should be a positive integer') ;
end
nV = numel(V) ;

if nargin==2
    %% PERMN(V,N) - return all permutations
```

```
    if nV == 0 || N == 0
        M = zeros(nV, N) ;
        I = zeros(nV, N) ;
    elseif N == 1
        % return column vectors
        M = V(:) ;
        I = (1:nV).' ;
    else
        % this is faster than the math trick used with 3 inputs below
        [Y{N:-1:1}] = ndgrid(1:nV) ;
        I = reshape(cat(N+1, Y{:}), [], N) ;
        M = V(I) ;
    end
else
    %% PERMN(V,N,K) - return a subset of all permutations
    nK = numel(K) ;
    if nV == 0 || N == 0 || nK == 0
        M = zeros(numel(K), N) ;
        I = zeros(numel(K), N) ;
    elseif nK < 1 || any(K<1) || any(K ~= fix(K))
        error('permn:InvalidIndex','Third argument should contain positive integers.') ;
    else
        V = reshape(V, 1, []) ; % v1.1 make input a row vector
        nV = numel(V) ;
        Npos = nV^N ;
        if any(K > Npos)
            warning('permn:IndexOverflow', ...
                'Values of K exceeding the total number of combinations are saturated.')
            K = min(K, Npos) ;
        end

        % The engine is based on version 3.2 with the correction
        % suggested by Roger Stafford. This approach uses a single matrix
        % multiplication.
        B = nV.^(1-N:0) ;
        I = ((K(:)-.5) * B) ; % matrix multiplication
        I = rem(floor(I), nV) + 1 ;
        M = V(I) ;
    end
end

% Algorithm using for-loops
% which can be implemented in C or VB
%
% nv = length(V) ;
% C = zeros(nv^N,N) ; % declaration
% for ii=1:N,
%     cc = 1 ;
%     for jj=1:(nv^(ii-1)),
%         for kk=1:nv,
%             for mm=1:(nv^(N-ii)),
%                 C(cc,ii) = V(kk) ;
%                 cc = cc + 1 ;
%             end
%         end
%     end
% end
```

B.6 removedups2 Code

The MATLAB code `removedups2` is used to create the orientation script to be run on the original limpet shell geometry, orienting it and saving all 102 new, oriented geometries.

```
function [D,L]=removedups2(choose)

a='ABCDE';
combosL=permn(a,choose);

b=[0 45 90 135 180];
combosD=permn(b,3);

R=zeros(length(combosD),9);
for i=1:length(combosD)

    alpha=combosD(i,3);
    beta=combosD(i,2);
    gamma=combosD(i,1);

    %%Yaw rotation matrix
    Rzi=[cosd(alpha) -sind(alpha) 0; sind(alpha) cosd(alpha) 0; 0 0 1];

    %%Pitch rotation matrix
    Ryi=[cosd(beta) 0 sind(beta); 0 1 0; -sind(beta) 0 cosd(beta)];

    %%Roll rotation matrix
    Rxi=[1 0 0; 0 cosd(gamma) -sind(gamma); 0 sind(gamma) cosd(gamma)];

    %%Unique rotation matrix
    Ri=Rzi*Ryi*Rxi;

    %Vectorize
    R(i,:)=Ri(:);
end

%Find unique rows of R
[U1,ia1]=unique(R,'rows','stable');
dupes = setdiff(1:size(U1,1),ia1);

%Use unique indices to print unique orientation combos
L=combosL(ia1,:);
D=combosD(ia1,:);

%Print duplicate orientation combos for checking
for i=1:length(dupes)
    dupesD(i,:)=combosD(dupes(:,i),:);
    dupesL(i,:)=combosL(dupes(:,i),:);
    W(i,:)=R(dupes(:,i),:);
end

fileID = fopen('duplicates.txt','w');
for i=1:length(R)
    [M]=ismember(R,R(i,:),'rows');
    dups=sum(M(:)==1)-1;
    if dups>=1
```

```
        X=find(M);
        N=combosL(X,:);
        j=1;
        while j<length(N)-1
            formatSpec=('%s is a duplicate of %s\n');
            fprintf(fileID,formatSpec,N(j,:),N(j+1:end,:));
            j=j+1;
        end
    end
end
fclose(fileID);

lines = strsplit(fileread('duplicates.txt'), '\n');  %read file and split into lines
lines = unique(lines, 'stable');  %remove duplicate lines
fid = fopen('duplicates.txt', 'w');    %open file for writing
fwrite(fid, strjoin(lines, '\n'), 'char'); %merge lines and write
fclose(fid);

%%writing script for orienting shell
fileID = fopen('orientscript.sh','w');
for i=1:length(D)
    formatSpec='(surfaceTransformPoints -rollPitchYaw ''(%d %d %d)'' flimpet.stl
     limp%s%s%s.stl) &\r\n';
    fprintf(fileID,formatSpec,D(i,1),D(i,2),D(i,3),L(i,1),L(i,2),L(i,3));
end
fclose(fileID);
```

B.7 translatescript Code

The MATLAB code `translatescript` is used to translate each new, oriented geometry to the appropriate location in the mesh domain.

```
format long

[~,L]=removedups2(3);

%%read .txt from surfaceCheck 'bounding box'
fid=fopen('bbox.txt');
tline = fgetl(fid);
tlines = cell(0,1);
while ischar(tline)
    tlines{end+1,1} = tline;
    tline = fgetl(fid);
end
tlines(2:2:end,:) = [];
tlines=split(tlines);
fclose(fid);

%%strip parentheses and make double
xmin=tlines(:,4);
for i=1:length(xmin)
    xmin{i}=strip(xmin{i},'left','(');
end
xmin=str2double(xmin);

xmax=tlines(:,7);
for i=1:length(xmax)
    xmax{i}=strip(xmax{i},'left','(');
end
xmax=str2double(xmax);

ymin=tlines(:,5);
ymin=str2double(ymin);

ymax=tlines(:,8);
ymax=str2double(ymax);

zmin=tlines(:,6);
for i=1:length(zmin)
    zmin{i}=strip(zmin{i},'right',')');
end
zmin=str2double(zmin);

zmax=tlines(:,9);
for i=1:length(zmax)
    zmax{i}=strip(zmax{i},'right',')');
end
zmax=str2double(zmax);

%%compute length, height, and span of geometries
for i=1:length(xmax)
    long(i)=xmax(i)-xmin(i);
end
```

```
long=long';

for i=1:length(ymax)
    height(i)=ymax(i)-ymin(i);
end
height=height';

for i=1:length(zmax)
    span(i)=zmax(i)-zmin(i);
end
span=span';
%
% %%find bilaterally symmetrical geometries
% xyz=[long height span];
%
% [C,ia]=unique(xyz,'rows','stable');
%
% combosL=combosL(ia,:);

%%compute needed translations
X=[];
for i=1:length(xmin)
    X(i,1)=0.15-xmin(i);
end

Y=[];
for i=1:length(ymin)
    Y(i,1)=0.002-ymin(i);
end

Z=[];
center=[];
for i=1:length(span)
    center(i,1)=(zmin(i)+zmax(i))/2;
    Z(i,1)=0.28-center(i);
end

% X=X(ia,:);
% Y=Y(ia,:);
% Z=Z(ia,:);

%%print into .sh script
fileID = fopen('translatescript.sh','w');
for i=1:length(L)
    formatSpec='(surfaceTransformPoints -translate ''(%d %d %d)'' limp%s%s%s.stl
    limp%s%s%s.stl) &\r\n';
    fprintf(fileID,formatSpec,X(i),Y(i),Z(i),L(i,1),L(i,2),L(i,3),L(i,1),L(i,2),L(i,3));
end
fclose(fileID);
```

Appendix C

OpenFOAM Dictionaries

C.1 blockMesh Dictionary

The OpenFOAM dictionary `blockMeshDict` creates a simple mesh out of 3-dimensional blocks.

```
 /*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v1912                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
convertToMeters 1;

vertices
(
    (0 0 0)
    (.72 0 0)
    (.72 .4  0)
    (0 .4 0)
    (0 0 .56)
    (.72 0 .56)
    (.72 .4 .56)
    (0 .4 .56)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (63 35 49) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
```

```
    frontAndBack
    {
        type patch;
        faces
        (
            (4 5 6 7)
            (0 3 2 1)
        );
    }
    inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (2 6 5 1)
        );
    }
    upperWall
    {
        type patch;
        faces
        (
            (3 7 6 2)
        );
    }
    lowerWall
    {
        type     wall;
        faces
        (
            (1 5 4 0)
        );
    }
);

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

C.2 snappyHexMesh Dictionary

The OpenFOAM dictionary `snappyHexMeshDict` chisels the mesh created by `blockMeshDict` into the desired mesh.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v1912                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      snappyHexMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       true;


// Geometry. Definition of all surfaces. All surfaces are of class
// searchableSurface.
// Surfaces are used
// - to specify refinement for any mesh cell intersecting it
// - to specify refinement for any mesh cell inside/outside/near
// - to 'snap' the mesh boundary to the surface
geometry
{
    flimpet.stl
    {
        type triSurfaceMesh;
        name limpet;
    }

    floorRefinement
    {
        type searchableBox;
        min  (0 0 0.18);
        max  (0.181023 0.00508 0.38);
    }

    wake1
    {
type searchableBox;
        min (0.181023 0 0.229821);
        max (0.310709 0.102358 0.330179);
    }

    wake2
```

```
    {
        type searchableBox;
        min (0.181023 0 0.229821);
        max (0.318681 0.102358 0.330179);
    }

    wake3
    {
        type searchableBox;
        min (0.181023 0 0.229821);
        max (0.329268 0.102358 0.330179);
    }
}


// Settings for the castellatedMesh generation.
castellatedMeshControls
{

    // Refinement parameters
    // ~~~~~~~~~~~~~~~~~~~~~

    // If local number of cells is >= maxLocalCells on any processor
    // switches from from refinement followed by balancing
    // (current method) to (weighted) balancing before refinement.
    maxLocalCells 150000000;

    // Overall cell limit (approximately). Refinement will stop immediately
    // upon reaching this number so a refinement level might not complete.
    // Note that this is the number of cells before removing the part which
    // is not 'visible' from the keepPoint. The final number of cells might
    // actually be a lot less.
    maxGlobalCells 200000000;

    // The surface refinement loop might spend lots of iterations refining just a
    // few cells. This setting will cause refinement to stop if <= minimumRefine
    // are selected for refinement. Note: it will at least do one iteration
    // (unless the number of cells to refine is 0)
    minRefinementCells 10;

    // Allow a certain level of imbalance during refining
    // (since balancing is quite expensive)
    // Expressed as fraction of perfect balance (= overall number of cells /
    // nProcs). 0=balance always.
    maxLoadUnbalance 0.10;


    // Number of buffer layers between different levels.
    // 1 means normal 2:1 refinement restriction, larger means slower
    // refinement.
    nCellsBetweenLevels 4;



    // Explicit feature edge refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies a level for any cell intersected by its edges.
```

```
    // This is a featureEdgeMesh, read from constant/triSurface for now.
    features
    (
    );



    // Surface based refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies two levels for every surface. The first is the minimum level,
    // every cell intersecting a surface gets refined up to the minimum level.
    // The second level is the maximum level. Cells that 'see' multiple
    // intersections where the intersections make an
    // angle > resolveFeatureAngle get refined up to the maximum level.

    refinementSurfaces
    {
        limpet
       {
            // Surface-wise min and max refinement level
           level (6 6);
        }

//floorRefinement
//{
//     level (6 6);
//}

//wakeRefinement
//{
//      level (6 6);
//}

     }

    // Resolve sharp angles
    resolveFeatureAngle 30;


    // Region-wise refinement
    // ~~~~~~~~~~~~~~~~~~~~~~

    // Specifies refinement level for cells in relation to a surface. One of
    // three modes
    // - distance. 'levels' specifies per distance to the surface the
    //   wanted refinement level. The distances need to be specified in
    //   descending order.
    // - inside. 'levels' is only one entry and only the level is used. All
    //   cells inside the surface get refined up to the level. The surface
    //   needs to be closed for this to be possible.
    // - outside. Same but cells outside.

    refinementRegions
    {
        floorRefinement
        {
            mode inside;
```

```
            levels ((1E15 5));
        }

        wake1
        {
            mode inside;
            levels ((1E15 5));
        }

wake2
        {
            mode inside;
            levels ((1E15 4));
        }

wake3
        {
            mode inside;
            levels ((1E15 3));
        }

//limpet
//{
// mode distance;
// levels ((1.0 5) (2.0 3));
//}
    }


    // Mesh selection
    // ~~~~~~~~~~~~~~

    // After refinement patches get added for all refinementSurfaces and
    // all cells intersecting the surfaces get put into these patches. The
    // section reachable from the locationInMesh is kept.
    // NOTE: This point should never be on a face, always inside a cell, even
    // after refinement.
    locationInMesh (0.44 0.15 0.26);


    // Whether any faceZones (as specified in the refinementSurfaces)
    // are only on the boundary of corresponding cellZones or also allow
    // free-standing zone faces. Not used if there are no faceZones.
    allowFreeStandingZoneFaces true;
}



// Settings for the snapping.
snapControls
{
    //- Number of patch smoothing iterations before finding correspondence
    //  to surface
    nSmoothPatch 5;

    //- Relative distance for points to be attracted by surface feature point
    //  or edge. True distance is this factor times local
    //  maximum edge length.
```

```
    tolerance 4.0;

    //- Number of mesh displacement relaxation iterations.
    nSolveIter 0;

    //- Maximum number of snapping relaxation iterations. Should stop
    //  before upon reaching a correct mesh.
    nRelaxIter 5;

    // Feature snapping

        //- Number of feature edge snapping iterations.
        //  Leave out altogether to disable.
        nFeatureSnapIter 10;

        //- Detect (geometric only) features by sampling the surface
        //  (default=false).
        implicitFeatureSnap false;

        //- Use castellatedMeshControls::features (default = true)
        explicitFeatureSnap true;

        //- Detect points on multiple surfaces (only for explicitFeatureSnap)
        multiRegionFeatureSnap false;
}



// Settings for the layer addition.
addLayersControls
{
    // Are the thickness parameters below relative to the undistorted
    // size of the refined cell outside layer (true) or absolute sizes (false).
    relativeSizes true;

    // Per final patch (so not geometry!) the layer information
    layers
    {
        "limpet*"
        {
            nSurfaceLayers 2;
        }
    }

    // Expansion factor for layer mesh
    expansionRatio 1.2;

    // Wanted thickness of final added cell layer. If multiple layers
    // is the thickness of the layer furthest away from the wall.
    // Relative to undistorted size of cell outside layer.
    // See relativeSizes parameter.
    finalLayerThickness 0.4;

    // Minimum thickness of cell layer. If for any reason layer
    // cannot be above minThickness do not add layer.
    // Relative to undistorted size of cell outside layer.
    minThickness 0.1;
```

```
    // If points get not extruded do nGrow layers of connected faces that are
    // also not grown. This helps convergence of the layer addition process
    // close to features.
    // Note: changed(corrected) w.r.t 1.7.x! (didn't do anything in 1.7.x)
    nGrow 0;

    // Advanced settings

    // When not to extrude surface. 0 is flat surface, 90 is when two faces
    // are perpendicular
    featureAngle 310;

    // At non-patched sides allow mesh to slip if extrusion direction makes
    // angle larger than slipFeatureAngle.
    slipFeatureAngle 30;

    // Maximum number of snapping relaxation iterations. Should stop
    // before upon reaching a correct mesh.
    nRelaxIter 3;

    // Number of smoothing iterations of surface normals
    nSmoothSurfaceNormals 1;

    // Number of smoothing iterations of interior mesh movement direction
    nSmoothNormals 3;

    // Smooth layer thickness over surface patches
    nSmoothThickness 10;

    // Stop layer growth on highly warped cells
    maxFaceThicknessRatio 0.5;

    // Reduce layer growth where ratio thickness to medial
    // distance is large
    maxThicknessToMedialRatio 0.3;

    // Angle used to pick up medial axis points
    // Note: changed(corrected) w.r.t 1.7.x! 90 degrees corresponds to 130
    // in 1.7.x.
    minMedialAxisAngle 80;


    // Create buffer region for new layer terminations
    nBufferCellsNoExtrude 0;


    // Overall max number of layer addition iterations. The mesher will exit
    // if it reaches this number of iterations; possibly with an illegal
    // mesh.
    nLayerIter 5000;
}



// Generic mesh quality settings. At any undoable phase these determine
// where to undo.
meshQualityControls
{
```

```
    #include "meshQualityDict"

}


// Advanced

// Write flags
writeFlags
(
    scalarLevels
    layerSets
    layerFields     // write volScalarField for layer coverage
);


// Merge tolerance. Is fraction of overall bounding box of initial mesh.
// Note: the write tolerance needs to be higher than this.
mergeTolerance 1e-6;


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

C.3 control Dictionary

The OpenFOAM dictionary `controlDict` specifies the main case controls, such as time steps, write format, etc.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | cfMesh: A library for mesh generation           |
|  \\    /   O peration     |                                                 |
|   \\  /    A nd           | Author: Franjo Juretic                          |
|    \\/     M anipulation  | E-mail: franjo.juretic@c-fields.com             |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version   2.0;
    format    ascii;
    class     dictionary;
    object    controlDict;
}
// ************************************************************************* //

application         simpleFoam;

startFrom           latestTime;

startTime           4;

stopAt              endTime;

endTime             3;

deltaT              0.002;

writeControl        timeStep;

writeInterval       1;

writeFormat         ascii;

runTimeModifiable   yes;

adjustTimeStep     on;

maxCo     0.1;

maxDeltaT     1;

functions
{
    forces
    {
        type            forceCoeffs;
        libs            (forces);
        writeControl    timeStep;
writeInterval 5;

        patches
```

```
        (
            limpet
        );

        p p;
U U;
rho rhoInf;
rhoInf 998;

porosity no;
writeFields yes;

        CofR        (0 0 0);
        liftDir     (0 1 0);
        dragDir     (1 0 0);
        pitchAxis   (0 0 1);
        magUInf     1.500000e-01;
        lRef        0.08;
        Aref        0.0105;
    }
    Co1
    {
type CourantNo;
libs ("libfieldFunctionObjects.so");
executeControl timeStep;
executeInterval 5;
writeControl writeTime;
    }

    vorticity
    {
     type            vorticity;
     libs            (fieldFunctionObjects);
        executeControl    timeStep;
        executeInterval    5;
        writeControl    writeTime;
    }
}


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

C.4 fvSolution Dictionary

The OpenFOAM dictionary `fvSolution` specifies the solver and algorithm being used by OpenFOAM.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | cfMesh: A library for mesh generation           |
|  \\    /   O peration     |                                                 |
|   \\  /    A nd           | Author: Franjo Juretic                          |
|    \\/     M anipulation  | E-mail: franjo.juretic@c-fields.com             |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver           GAMG;
        tolerance        1e-06;
        relTol           0.1;
smoother  GaussSeidel;
    }

    "(U|k|epsilon|omega|f|v2)"
    {
        solver           smoothSolver;
smoother  symGaussSeidel;
        tolerance        1e-08;
        relTol           0.1;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;
    consistent      yes;

    residualControl
    {
      p                             1e-9;
      U                             1e-9;
      "(k|epsilon|omega|f|v2)"      1e-9;
    }

}

relaxationFactors
{
```

```
    equations
    {
U 0.7;
".*" 0.5;
    }
}


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

C.5 fvSchemes Dictionary

The OpenFOAM dictionary `fvSchemes` specifies the numerical schemes being used by OpenFOAM.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss linearUpwind grad(U);
    div(phi,k)      bounded Gauss limitedLinear 1;
    div(phi,epsilon) bounded Gauss limitedLinear 1;
    div(phi,omega)  bounded Gauss limitedLinear 1;
    div(phi,v2)     bounded Gauss limitedLinear 1;
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
    div(nonlinearStress) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
    interpolate(U)  linear;
}

snGradSchemes
{
```

```
    default         corrected;
}

wallDist
{
    method meshWave;
}


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

C.6 Velocity Dictionary

The OpenFOAM dictionary `U` sets the velocity boundary conditions to be used in simulation.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions [0 1 -1 0 0 0 0];
internalField uniform (0.15 0 0);

boundaryField
{
    inlet
    {
        type            fixedProfile;
        profile         csvFile;
        profileCoeffs
        {
                nHeaderLine             0;
                refColumn               0;
                componentColumns        3(1 2 3);
                separator               ",";
                mergeSeparators         0;
                file                    "0/logUP.csv";
        }
        direction       (0 1 0);
        origin          0;
    }

    outlet
    {
        type            inletOutlet;
        inletValue uniform (0 0 0);
value $internalField;
    }

    frontAndBack
    {
        type           slip;
    }

    lowerWall
    {
type            noSlip;
```

```
    }
    upperWall
    {
        type            slip;
    }
    "limpet*"
    {
        type            fixedValue;
value uniform (0 0 0);
    }
}

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

C.7 Turbulent Kinetic Energy Dictionary

The OpenFOAM dictionary `k` sets the turbulent kinetic energy k boundary conditions to be used in simulation.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      k;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions [0 2 -2 0 0 0 0];

internalField   uniform 1.137904e-04;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value $internalField;
    }
    outlet
    {
        type            inletOutlet;
inletValue $internalField;
value $internalField;
    }
    frontAndBack
    {
        type slip;
    }
    lowerWall
    {
        type            fixedValue;
value uniform 0;
    }
    upperWall
    {
        type            slip;
    }
    "limpet*"
    {
        type            fixedValue;
   value           uniform 0;
```

```
    }
}

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

C.8 Turbulence Specific Dissipation Rate Dictionary

The OpenFOAM dictionary `omega` sets the turbulent specific dissipation rate ω boundary conditions to be used in simulation.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      omega;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions [0 0 -1 0 0 0 0];

internalField   uniform 1.217229e-01;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }
    outlet
    {
        type            inletOutlet;
inletValue $internalField;
value $internalField;
    }
    lowerWall
    {
        type            fixedValue;
        value           uniform 10^8;
    }
    frontAndBack
    {
        type            slip;
    }
    upperWall
    {
        type            slip;
    }
    "limpet*"
    {
        type            fixedValue;
value uniform 10^8;
```

```
    }
}


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

C.9 Turbulent Viscosity Dictionary

The OpenFOAM dictionary `nut` sets the turbulent viscosity ν_t boundary conditions to be used in simulation.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      nut;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions [0 2 -1 0 0 0 0];

internalField   uniform 1.662392e-03;

boundaryField
{
    inlet
    {
        type            calculated;
        value           uniform 0;
    }
    outlet
    {
        type            calculated;
value uniform 0;
    }
    frontAndBack
    {
        type            calculated;
value uniform 0;
    }
    lowerWall
    {
        type            fixedValue;
value uniform 0;
    }
    upperWall
    {
        type            calculated;
value uniform 0;
    }
    "limpet*"
    {
        type            fixedValue;
```

```
value uniform 0;
    }
}


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

C.10 Pressure Dictionary

The OpenFOAM dictionary p sets the pressure p boundary conditions to be used in simulation.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            fixedValue;
        value           $internalField;
    }

    lowerWall
    {
        type           zeroGradient;
    }

    frontAndBack
    {
        type           slip;
    }
    upperWall
    {
        type           slip;
    }
    "limpet*"
    {
        type            zeroGradient;
    }

}
```

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

BIBLIOGRAPHY

[1] K. A. Ahmad, W. McEwan, J. K. Watterson, and J. Cole. Rans turbulence models for pitching airfoil. *WIT Transactions on The Built Environment*, 84, 2005.

[2] S. V. Apte, K. Mahesh, and T. Lundgren. Accounting for finite-size effects in simulations of disperse particle-laden flows. *International Journal for Multiphase Flow*, 34:260–271, 2008.

[3] O. Ayala, W. Grabowski, and L. Wang. A hybrid approach for simulating turbulent collisions of hydrodynamically-interacting particles. *Journal of Computational Physics*, 225:51–73, 2007.

[4] J. Bredberg. Sst k-omega model, 2005.

[5] P. Dellino, D. Mele, R. Bonasia, G. Braia, L. Volpe, and R. Sulpizio. The influence of pumice shape on its terminal velocity. *Geophysical Research Letters: Solid Earth*, 32(21), 2005.

[6] M. Denny. A limpet shell shape that reduces drag: laboratory demonstration of a hydrodynamic mechanism and an exploration of its effectiveness in nature. *Canadian Journal of Zoology*, 67:2098–2106, 1989.

[7] F. Dioguardi, D. Mele, and P. Dellino. A new one-equation model of fluid drag for irregularly shaped particles valid over a wide range of reynolds number. *Journal of Geophysical Research: Solid Earth*, 123(144-156), 2018.

[8] E. Hauf. Gastropod: Patella sp. 3D Model, Paleontological Research Institution.

[9] A. Joelchorin. Numerical study of slightly viscous flow. *Computational Wind Engineering*, 1, 1993.

[10] D. Leith. Drag on nonspherical objects. *Aerosol Science and Technology*, 6(2):153–161, 1987.

[11] X. Ortiz, D. Rival, and D. Wood. Forces and moments on flat plates of small aspect ratio with application to pv wind loads and small wind turbine blades. *Energies*, 8:2438–2453, 2015.

[12] Proceedings of the ASME 2015 34th International Conference on Ocean, Offshore and Arctic Engineering. *The Pros and Cons of Wall Functions*, volume 2: CFD and VIV, 2015.

[13] E. Shams, J. Finn, and S. V. Apte. A numerical scheme for euler-lagrange simulation of bubbly flows in complex systems. *International Journal for Numerical Methods in Fluids*, 1(1), 2010.

[14] SimScale. K-omega and k-omega sst.

[15] C. N. A. Troch. *Incipient motion of shells and shell fragments*. PhD thesis, Stellenbosch University, December 2015.