fluids

Tutorial

A CFD Tutorial in Julia: Introduction to Compressible Laminar Boundary-Layer Flows

Furkan Oz * and Kursat Kara

School of Mechanical and Aerospace Engineering, Oklahoma State University, Stillwater, OK 74078, USA; kursat.kara@okstate.edu
* Correspondence: foz@okstate.edu

Abstract: A boundary-layer is a thin fluid layer near a solid surface, and viscous effects dominate it. The laminar boundary-layer calculations appear in many aerodynamics problems, including skin friction drag, flow separation, and aerodynamic heating. A student must understand the flow physics and the numerical implementation to conduct successful simulations in advanced undergraduate- and graduate-level fluid dynamics/aerodynamics courses. Numerical simulations require writing computer codes. Therefore, choosing a fast and user-friendly programming language is essential to reduce code development and simulation times. Julia is a new programming language that combines performance and productivity. The present study derived the compressible Blasius equations from Navier–Stokes equations and numerically solved the resulting equations using the Julia programming language. The fourth-order Runge–Kutta method is used for the numerical discretization, and Newton's iteration method is employed to calculate the missing boundary condition. In addition, Burgers', heat, and compressible Blasius equations are solved both in Julia and MATLAB. The runtime comparison showed that Julia with *for* loops is 2.5 to 120 times faster than MATLAB. We also released the Julia codes on our GitHub page to shorten the learning curve for interested readers.

Keywords: CFD; boundary-layer; compressible flow; Julia; MATLAB; similarity solution

Citation: Oz, F.; Kara, K. A CFD Tutorial in Julia: Introduction to Compressible Laminar Boundary-Layer Flows. *Fluids* **2021**, *6*, 400. https://doi.org/10.3390/fluids6110400

Academic Editors: Laura A. Miller, Nicholas Battista, Amy Buchmann and Antonis Anastasiou

Received: 13 October 2021
Accepted: 2 November 2021
Published: 5 November 2021

1. Introduction

Until the 19th century, scientists neglected the effects of viscosity in their hydrodynamic and aerodynamic calculations using potential flow theory. However, this assumption led to a contradiction between theoretical predictions and experimental measurements of drag force acting on a moving body, now known as the d'Alembert paradox [1]. Later, a revolutionary boundary-layer concept is introduced [2,3]. In this concept, the fluid flow over a surface is divided into two regions by the boundary-layer edge: an area between the surface and the boundary-layer edge dominated by the viscous effects and a region outside the boundary-layer edge where the viscous effects can be neglected. It enables a significant simplification of full Navier–Stokes equations.

The boundary-layer theory was first presented by Prandtl [4] in 1904, and it provides the solutions of velocity and temperature profiles within the boundary-layer by using approximations. One can obtain Blasius [5,6], Falkner–Skan [7], and compressible Falkner–Skan [2,8] solutions by using this approach. Researchers extensively use these solutions to validate the computational fluid dynamics (CFD) simulations. Moreover, in a CFD simulation, one can calculate the boundary-layer thickness in advance to estimate the required grid parameters to resolve the boundary-layer region. Understanding the fundamentals of boundary-layer theory is critical for engineers to solve today's aerodynamic design challenges.

It may be challenging to fully understand the fundamentals of the boundary-layer theory in undergraduate- and graduate-level boundary-layer courses. Most of the time, books skip or briefly mention some steps in the derivation of a system of equations.

Additionally, instructors are forced to leave the details of derivations to students due to the limited lecture time. The steps that are skipped may become a challenge for students. Moreover, the derived equations usually do not have an analytical solution; therefore, they must be solved using numerical methods. Students or engineers who do not have adequate experience in the subject may struggle to understand the details of the topics because of the blanks in the process. A tutorial of step-by-step derivation and implementation in the computer environment may help students to fill the blanks. Moreover, researchers from another field may utilize the code and/or the simple explanation of the topic in their research. For the coding part, there are several available coding languages extensively used in scientific community, such as Fortran [9], Python [10], C/C++ [11], and MATLAB [12]. However, Julia [13] may be another alternative for students to write high-level, generic code that resembles mathematical formulas. It is a relatively new, fast, and dynamic coding language that focuses on productivity. It is trying to fill the gap between high-performance languages, such as Fortran and C/C++, and user-friendly languages, such as Python and MATLAB. Students tend to use user-friendly languages for their coursework and simple problems; however, in the industry, it is crucial to have a fast solver. In this gap, Julia provides easy syntax, as Python and MATLAB, and a fast performance, as Fortran and C/C++. This makes Julia a great choice for researchers due to the ability to combine high-performance with productivity. Although there are some tutorial papers and modules developed in other languages [14–17], the current number of publications is not enough to gain a thorough understanding of the Julia language in CFD [18,19].

In this tutorial paper, compressible Blasius equation and energy equation are derived from scratch and implemented in the Julia environment. The fourth-order Runge–Kutta method is employed to solve the final differential equations and Newton's iteration method is used for the missing boundary condition. Solutions obtained by the code are validated with Iyer's [20] BL2D boundary-layer solver which is used in NASA's well-known compressible boundary-layer stability solver Langley Stability and Transition Analysis Code (LASTRAC) [21]. The derivation details with the numerical implementation will guide students to understand the compressible laminar boundary-layer concept better. It will be easier for them to solve more complex problems with their own codes. Figure 1 illustrates the visual abstract of the paper, which gives the main ideas of the present paper. Additionally, Burger's, heat, and compressible Blasius equations solution times obtained by Julia and MATLAB solvers are compared with each other. We make all these codes available on GitHub to shorten the learning curve. We provide the GitHub link of the codes, installation instructions, and required packages in Appendix A. Advanced boundary-layer topics are beyond the scope of this paper, interested readers are referred to additional references [22–25] for subsonic boundary-layer transition, references [26–34] for supersonic/hypersonic boundary-layer transition, and references [35–37] for flow separation. The other research studies where boundary-layer flow is involved are presented in the references [38–43].

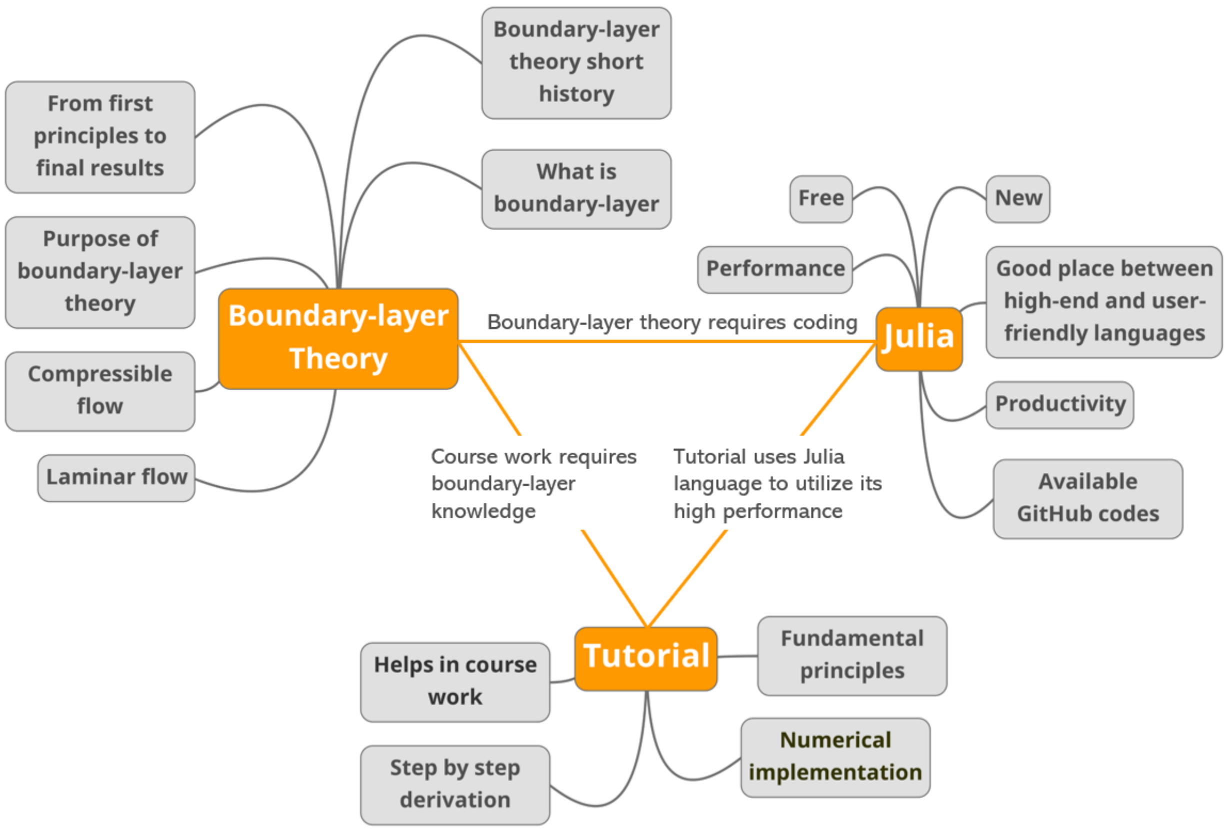

Boundary-layer Theory
From first principles to final results
Purpose of boundary-layer theory
Compressible flow
Laminar flow
Boundary-layer theory short history
What is boundary-layer
Boundary-layer theory requires coding
Julia
Free
New
Performance
Good place between high-end and user-friendly languages
Productivity
Available GitHub codes
Course work requires boundary-layer knowledge
Tutorial uses Julia language to utilize its high performance
Tutorial
Helps in course work
Step by step derivation
Fundamental principles
Numerical implementation

Figure 1. The visual abstract of the present paper which is mainly designed around 3 major points. Major points are further divided into smaller points which correspond to purposes/ideas of the particular major point. Each major point is connected to one another, which makes them complete.

2. Compressible Laminar Boundary-Layer

Compressibility effect in the boundary-layer requires additional calculations. Constant density assumption in incompressible speeds is no longer valid for the compressible boundary-layer. In compressible speeds, temperature and density change within the boundary-layer. It is crucial to capture the velocity, temperature and density variations in the boundary-layer to obtain accurate simulation results. One can estimate the number of element required to resolve the boundary-layer in the CFD simulation by using the boundary-layer theory. Compressible Blasius is also widely used for CFD validations in high-speed flows. In this section, compressible Blasius equations will be derived from scratch and implemented in the Julia environment. The contribution of this paper is employing the Julia language. The equations used in this paper are already in the literature [2,44]. The manuscript may enable students to adopt the programming language with easily and available GitHub codes, which may shorten the learning curve.

2.1. *Compressible Blasius Equations*

Incompressible Blasius solution is a similarity solution for a flat plate. The assumptions for the incompressible Blasius equations are given in our previous work [19]; interested readers can check the details from there. In the compressible region, the temperature effects must be taken into account for an accurate solution. In the incompressible region, the temperature and density changes are small enough to be neglected. In the compressible region, the temperature can increase drastically as a result; density decreases within the boundary-layer. For example, the temperature on the solid wall can reach 7 times the freestream temperature in Mach 6 flow over a wedge. If the freestream temperature is 300 K, the wall temperature will be around 2100 K. In order to compare the quantity, the melting point of titanium is around 1941 K [45]. This problem is still a challenge for aerospace applications in which high Mach numbers are involved.

The compressible Blasius equations can be derived from the compressible Navier–Stokes equations, which can be expressed in two spatial dimensions as:

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} = 0 \tag{1}$$

$$\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) = -\frac{\partial p}{\partial x} + \frac{\partial}{\partial x}\left[2\mu\frac{\partial u}{\partial x} + \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right] + \frac{\partial}{\partial y}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right] \tag{2}$$

$$\rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) = -\frac{\partial p}{\partial y} + \frac{\partial}{\partial x}\left[\mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right)\right] + \frac{\partial}{\partial y}\left[2\mu\frac{\partial v}{\partial y} + \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right] \tag{3}$$

$$\rho c_p\left(\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}\right) = -u\frac{\partial p}{\partial x} - v\frac{\partial p}{\partial y} + \frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right) + \frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right) + \Phi, \tag{4}$$

where ρ is the density, u and v are the velocities in x- and y- directions, p is the pressure, μ is the dynamic viscosity, λ is the second viscosity coefficient, k is the thermal conductivity, T is the temperature, c_p is the specific heat at constant pressure, and Φ is the dissipation function, which can be written as:

$$\Phi = \mu\left[2\left(\frac{\partial u}{\partial x}\right)^2 + 2\left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)^2\right] + \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)^2. \tag{5}$$

In order to obtain the boundary-layer equations, dimensional analysis is required to neglect the variables that have smaller orders than others. The flat plate boundary-layer development is illustrated in Figure 2. In this flow, u velocity is related to freestream velocity and the order of magnitude is one. The x is related to plate length, so its order of magnitude is also one. The y distance is related to boundary-layer thickness, so it is in the order of δ which is the boundary-layer thickness. The density, ρ, is related to freestream density so its order of magnitude is also one. The magnitude of the v velocity can be calculated from the continuity equation, Equation (1). In order to get zero from this equation, all variables must be in the same order so v is in the order of δ as a result of this, $\frac{\partial(\rho v)}{\partial y} = O(1)$. When the magnitude analysis is completed in the same manner, the boundary-layer equations can be obtained. It has to be noted that dynamic viscosity is in the order of δ^2, pressure and temperature are in the order of one. The specific heat at constant pressure is in the order of one. The second viscosity coefficient, λ, can be taken as $-2/3\mu$ because of Stokes' hypothesis. Once the order of magnitude is obtained for each of the terms, some of the terms can be neglected because $\delta \ll 1$. The final system of equations in steady-state condition ($\frac{\partial}{\partial t} = 0$) will be:

$$\frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} = 0 \tag{6}$$

$$\rho\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) = -\frac{\partial p}{\partial x} + \frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial y}\right) \tag{7}$$

$$\frac{\partial p}{\partial y} = 0 \tag{8}$$

$$\rho c_p\left(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}\right) = -u\frac{\partial p}{\partial x} + \frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right) + \mu\left(\frac{\partial u}{\partial y}\right)^2. \tag{9}$$

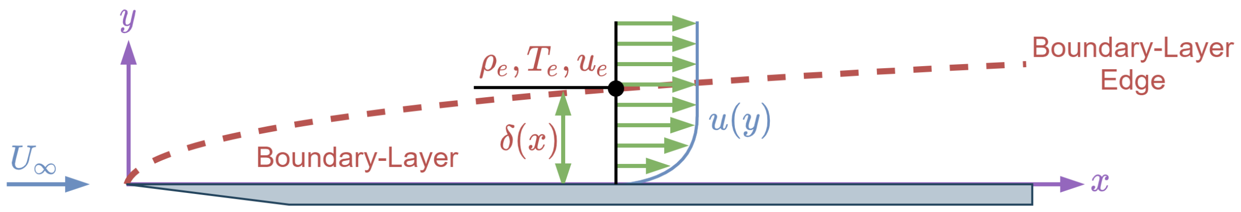


Figure 2. Schematic description of the flow over a flat plate. The red dashed line corresponds to boundary-layer edge. The boundary-layer velocity profile is illustrated with a blue line. The black dot corresponds to the boundary-layer edge at that station. The density, temperature, and velocity at the boundary-layer edge are ρ_e, T_e, and u_e, respectively. The boundary-layer thickness is defined with $\delta(x)$, which is the function of x.

Equation (7) can be expressed at the boundary-layer edge as:

$$\rho u_e \frac{\partial u_e}{\partial x} = -\frac{\partial p_e}{\partial x}. \tag{10}$$

The variables are changing from the solid surface up to the boundary-layer edge. At the boundary-layer edge, they reach to freestream value for the corresponding variable and remain constant. The velocity change in the y-direction at the boundary-layer edge is zero ($\frac{\partial u}{\partial y}|_{y=\delta} = 0$), because it is constant at boundary-layer edge. Equation (8) indicates that the pressure gradient in the y-direction is zero, so pressure at the boundary-layer edge equals the pressure within the boundary-layer ($p_e = p$). Equation (10) becomes:

$$\rho u_e \frac{\partial u_e}{\partial x} = -\frac{\partial p}{\partial x}. \tag{11}$$

The velocity at the boundary-layer edge is equal to freestream velocity, which is constant in x-direction for a flat plate. In other words, edge velocity gradient in x-direction is zero ($\frac{\partial u_e}{\partial x} = 0$). If the equation of state is used to obtain the ratio of density and temperature as:

$$p = \rho R T \tag{12}$$

$$p_e = \rho_e R T_e, \tag{13}$$

where R is the gas constant. It is known that $p = p_e$, so $\rho T = \rho_e T_e$. The final system of equations is:

$$\frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} = 0 \tag{14}$$

$$\rho\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) = \frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial y}\right) \tag{15}$$

$$\frac{\partial p}{\partial y} = 0 \tag{16}$$

$$\rho c_p\left(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}\right) = \frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right) + \mu\left(\frac{\partial u}{\partial y}\right)^2, \tag{17}$$

where $\frac{T_e}{T} = \frac{\rho}{\rho_e}$. At this point, a similarity parameter can be introduced to the system to obtain a similarity solution [46]. The similarity parameter, η, can be defined as:

$$\eta = \frac{u_e \rho_e}{\sqrt{2s}} \int_0^y \frac{T_e}{T} dy, \tag{18}$$

where $s = \mu_e \rho_e u_e x$. Let's assume that the stream function is

$$\psi = \sqrt{2s} f(\eta). \tag{19}$$

The u and v velocities can be calculated from the stream function as:

$$u = \frac{1}{\rho}\frac{\partial \psi}{\partial y}, \quad v = -\frac{1}{\rho}\frac{\partial \psi}{\partial x} \tag{20}$$

In this step, the variables in Equation (15), u, v, $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, and $\frac{\partial}{\partial y}\left(\mu \frac{\partial u}{\partial y}\right)$ can be calculated. The first derivative of η with respect to y and the first derivative of s with respect to x will be required for the chain rule.

$$s = \mu_e \rho_e u_e x \tag{21}$$

$$\frac{ds}{dx} = \mu_e \rho_e u_e \tag{22}$$

$$\eta = \frac{u_e \rho_e}{\sqrt{2s}} \int_0^y \frac{T_e}{T} dy \tag{23}$$

$$\frac{\partial \eta}{\partial y} = \frac{u_e \rho}{\sqrt{2s}}. \tag{24}$$

It is better to note that, in $\frac{\partial \eta}{\partial y}$ calculation, $\frac{T_e}{T} = \frac{\rho}{\rho_e}$ relation is used. The u velocity can be calculated as:

$$u = \frac{1}{\rho}\frac{\partial \psi}{\partial y} \tag{25}$$

$$= \frac{1}{\rho}\frac{\partial \psi}{\partial \eta}\frac{\partial \eta}{\partial y} \tag{26}$$

$$= \frac{1}{\rho}\left(\sqrt{2s}\frac{df}{d\eta}\right)\frac{u_e \rho}{\sqrt{2s}} \tag{27}$$

$$= f' u_e \tag{28}$$

The same procedure can be applied for v velocity as:

$$v = -\frac{1}{\rho}\frac{\partial \psi}{\partial x} \tag{29}$$

$$= -\frac{1}{\rho}\left(\frac{\partial \psi}{\partial s}\frac{\partial s}{\partial x} + \frac{\partial \psi}{\partial \eta}\frac{\partial \eta}{\partial x}\right) \tag{30}$$

$$= -\frac{1}{\rho}\left[\left(\frac{1}{2\sqrt{2s}} 2f\right)(\mu_e \rho_e u_e) + \left(\sqrt{2s} f'\right)\left(\frac{\partial \eta}{\partial x}\right)\right] \tag{31}$$

$$= -\frac{1}{\rho}\left[\left(\frac{1}{\sqrt{2s}} f \mu_e \rho_e u_e\right) + \left(\sqrt{2s} f' \frac{\partial \eta}{\partial x}\right)\right]. \tag{32}$$

Once u velocity is obtained, the derivatives with respect to x and y can be calculated as:

$$\frac{\partial u}{\partial y} = \frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial y} \tag{33}$$

$$= \frac{u_e^2 \rho}{\sqrt{2s}} f'' \tag{34}$$

$$\frac{\partial}{\partial y}\left(\mu \frac{\partial u}{\partial y}\right) = \frac{\partial}{\partial \eta}\left(\mu \frac{\partial u}{\partial y}\right)\frac{\partial \eta}{\partial y} \tag{35}$$

$$= \frac{\partial}{\partial \eta}\left(\mu \frac{u_e^2 \rho}{\sqrt{2s}} f''\right)\frac{u_e \rho}{\sqrt{2s}} \tag{36}$$

$$= \frac{\partial}{\partial \eta}\left(\mu \rho f''\right)\frac{u_e^3 \rho}{2s} \tag{37}$$

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial x} \tag{38}$$

$$= \frac{\partial (u_e f')}{\partial \eta}\frac{\partial \eta}{\partial x} \tag{39}$$

$$= u_e f'' \frac{\partial \eta}{\partial x}. \tag{40}$$

All terms in Equation (15) are known. If the above terms are substituted into Equation (15) and the necessary simplifications are done, the final equation will be:

$$\frac{\partial}{\partial \eta}\left(f'' \frac{\rho}{\rho_e}\frac{\mu}{\mu_e}\right) + f f'' = 0. \tag{41}$$

It has to be noted that if $\rho = \rho_e$ and $\mu = \mu_e$, in other words, if the flow is incompressible, Equation (41) becomes an incompressible Blasius equation ($f''' + f f'' = 0$). Equation (41) can be further simplified as:

$$\frac{\rho}{\rho_e}\frac{\mu}{\mu_e} f''' + f'' \frac{\partial}{\partial \eta}\left(\frac{\rho}{\rho_e}\frac{\mu}{\mu_e}\right) + f'' f = 0 \tag{42}$$

$$f''' + \frac{\bar{\rho}}{\bar{\mu}}\frac{\partial}{\partial \eta}\left(\frac{\bar{\mu}}{\bar{\rho}}\right) f'' + \frac{\bar{\rho}}{\bar{\mu}} f f'' = 0, \tag{43}$$

where $\bar{\mu} = \frac{\mu}{\mu_e}$ and $\bar{\rho} = \frac{\rho_e}{\rho} = \frac{T}{T_e}$. The momentum equation of the compressible Blasius equations is obtained in Equation (42). The energy equation of the compressible Blasius equations can be obtained with the same procedure. First of all, $\frac{\partial T}{\partial x}$, $\frac{\partial T}{\partial y}$, and $\frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right)$ have to be calculated. These terms can be calculated as:

$$\frac{\partial T}{\partial x} = \frac{\partial T}{\partial \eta}\frac{\partial \eta}{\partial x} \tag{44}$$

$$= T_e \bar{\rho}' \frac{\partial \eta}{\partial x} \tag{45}$$

$$\frac{\partial T}{\partial x} = \frac{\partial T}{\partial \eta}\frac{\partial \eta}{\partial y} \tag{46}$$

$$= T_e \bar{\rho}' \frac{u_e \rho}{\sqrt{2s}} \tag{47}$$

$$\frac{\partial}{\partial y}\left(k \frac{\partial T}{\partial y}\right) = \frac{\partial}{\partial \eta}\left(k \frac{\partial T}{\partial y}\right)\frac{\partial \eta}{\partial y} \tag{48}$$

$$= \frac{\partial}{\partial \eta}\left(k T_e \bar{\rho}' \frac{u_e \rho}{\sqrt{2s}}\right)\frac{u_e \rho}{\sqrt{2s}} \tag{49}$$

$$= \frac{T_e u_e^2 \rho}{2s}\frac{\partial (k \rho \bar{\rho})}{\partial \eta}. \tag{50}$$

When these terms are substituted into Equation (17), the new equation will be:

$$\rho c_p(u_e f')\left(T_e\rho'\frac{\partial\eta}{\partial x}\right)+\rho c_p\frac{-1}{\rho}\left(\frac{1}{\sqrt{2s}}f\mu_e\rho_e u_e+\sqrt{2s}f'\frac{\partial\eta}{\partial x}\right)\left(T_e\bar{\rho}'\frac{u_e\rho}{\sqrt{2s}}\right)$$
$$=\frac{T_e u_e^2\rho}{2s}\frac{\partial(k\rho\bar{\rho}')}{\partial\eta}+\mu\left(\frac{\rho u_e^2}{\sqrt{2s}}f''\right)^2. \quad (51)$$

Equation (51) can be simplified by dividing it with $\rho\mu c_p$, substituting Prandtl number into the equation where Prandtl number $Pr=\frac{c_p\mu}{k}$ and multiplying with $\frac{Pr\bar{\rho}}{\bar{\mu}}$. The final equation will be:

$$\bar{\rho}''+\frac{\bar{\rho}}{\bar{\mu}}\bar{\rho}'\frac{\partial}{\partial\eta}\left(\frac{\bar{\mu}}{\bar{\rho}}\right)+\frac{Pr}{\bar{\mu}}\bar{\rho}f\bar{\rho}'+(\gamma-1)PrM_e^2f''^2=0, \quad (52)$$

where $c_p=\frac{\gamma}{\gamma-1}R$, $M=\frac{u_e}{a_e}$, and $a_e=\sqrt{\gamma RT_e}$. In the final system of equations, the $\bar{\mu}$ can be calculated from Sutherland Viscosity Law [47]. The dimensional viscosity function is:

$$\mu=\frac{c_1T^{3/2}}{T+c_2}, \quad (53)$$

where $c_1=1.458\times10^{-6}\frac{kg}{ms\sqrt{K}}$ and $c_2=110.4$ K. The $\bar{\mu}$ is:

$$\bar{\mu}=\frac{c_1T^{3/2}}{T+c_2}\frac{T_e+c_2}{c_1T_e^{3/2}} \quad (54)$$
$$=\left(\frac{T}{T_e}\right)^{3/2}\frac{1+\frac{c_2}{T_e}}{\frac{T}{T_e}+\frac{c_2}{T_e}} \quad (55)$$
$$=\bar{\rho}^{3/2}\frac{1+\frac{c_2}{T_e}}{\bar{\rho}+\frac{c_2}{T_e}}. \quad (56)$$

The derivative of the viscosity is also required. The derivative terms can be calculated as:

$$\frac{\partial}{\partial\eta}\left(\frac{\bar{\mu}}{\bar{\rho}}\right)=\left(1+\frac{c_2}{T_e}\right)\left[\frac{\frac{\bar{\rho}}{2\bar{\rho}^{1/2}}}{\bar{\rho}+\frac{c_2}{T_e}}-\frac{\bar{\rho}'\bar{\rho}^{3/2}}{\left(\bar{\rho}+\frac{c_2}{T_e}\right)^2}\right]. \quad (57)$$

The final system of equations is:

$$f'''+\frac{\bar{\rho}}{\bar{\mu}}\frac{\partial}{\partial\eta}\left(\frac{\bar{\mu}}{\bar{\rho}}\right)f''+\frac{\bar{\rho}}{\bar{\mu}}ff''=0 \quad (58)$$
$$\bar{\rho}''+\bar{\rho}'\frac{\bar{\rho}}{\bar{\mu}}\frac{\partial}{\partial\eta}\left(\frac{\bar{\mu}}{\bar{\rho}}\right)+Pr\frac{\bar{\rho}}{\bar{\mu}}f\bar{\rho}'+(\gamma-1)PrM_e^2f''^2=0. \quad (59)$$

It has to be emphasized that $\bar{\rho}$ is a function of η and the final system of equations is coupled, so they have to be solved together. The boundary conditions of the system for an adiabatic system are:

$$\eta\to\infty \quad f'=1 \quad (60)$$
$$\eta=0 \quad f=f'=0 \quad (61)$$
$$\eta\to\infty \quad \bar{\rho}=1 \quad (62)$$
$$\eta=0 \quad \bar{\rho}'=0. \quad (63)$$

The boundary condition for the isothermal wall depends on the wall temperature. For example, if the wall temperature equals the boundary-layer edge temperature, it will be $\bar{\rho} = 1$, and it will be replaced with the last boundary condition of the system. In the adiabatic boundary condition, the derivative of the temperature with respect to wall-normal direction will be 0. During the numerical procedures, the difference will be emphasized one more time.

2.2. Numerical Procedure

In this section, the compressible Blasius equation will be solved with the fourth-order Runge–Kutta method [48] and Newton's iteration method [49]. Different methods can be used for this problem; however, we used Runge–Kutta and Newton's method because of their extensive usage in the literature and accuracy. To start the numerical procedure, high-order differential equations can be reduced to the first-order differential equations as:

$$f = y_1 \tag{64}$$
$$f' = y_2 \tag{65}$$
$$f'' = y_3 \tag{66}$$
$$\bar{\rho} = y_4 \tag{67}$$
$$\bar{\rho}' = y_5 \tag{68}$$

if Equations (64)–(68) are substituted into Equations (58) and (59), the final version of these equations can be written as:

$$f''' = -y_3\left(\frac{y_5}{2y_4} - \frac{y_5}{y_4 + \frac{c_2}{T_e}}\right) - y_1 y_3\left(\frac{y_4 + \frac{c_2}{T_e}}{\sqrt{y_4}(1 + \frac{c_2}{T_e})}\right) \tag{69}$$
$$\bar{\rho}'' = -y_5^2\left(\frac{1}{2y_4} - \frac{1}{y_4 + \frac{c_2}{T_e}}\right) - Pr\frac{y_1 y_5}{\sqrt{y_4}}\frac{y_4 + \frac{c_2}{T_e}}{1 + \frac{c_2}{T_e}} - (\gamma - 1)PrM_e^2 y_3^2. \tag{70}$$

The final system of equations can be written in the matrix form as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}' = \begin{bmatrix} y_2 \\ y_3 \\ -y_3\left(\frac{y_5}{2y_4} - \frac{y_5}{y_4+\frac{c_2}{T_e}}\right) - y_1 y_3\left(\frac{y_4+\frac{c_2}{T_e}}{\sqrt{y_4}(1+\frac{c_2}{T_e})}\right) \\ y_5 \\ -y_5^2\left(\frac{1}{2y_4} - \frac{1}{y_4+\frac{c_2}{T_e}}\right) - Pr\frac{y_1 y_5}{\sqrt{y_4}}\frac{y_4+\frac{c_2}{T_e}}{1+\frac{c_2}{T_e}} - (\gamma-1)PrM_e^2 y_3^2 \end{bmatrix}. \tag{71}$$

The adiabatic boundary conditions for the system are:

$$f(\eta = 0) = 0 \Rightarrow y_1(\eta = 0) = 0 \tag{72}$$
$$f'(\eta = 0) = 0 \Rightarrow y_2(\eta = 0) = 0 \tag{73}$$
$$\bar{\rho}'(\eta = 0) = 0 \Rightarrow y_5(\eta = 0) = 0 \tag{74}$$
$$f'(\eta \to \infty) = 1 \Rightarrow y_2(\eta \to \infty) = 1 \tag{75}$$
$$\bar{\rho}(\eta \to \infty) = 1 \Rightarrow y_4(\eta \to \infty) = 1. \tag{76}$$

The isothermal boundary conditions for the system are:

$$f(\eta = 0) = 0 \quad \Rightarrow y_1(\eta = 0) = 0 \tag{77}$$

$$f'(\eta = 0) = 0 \quad \Rightarrow y_2(\eta = 0) = 0 \tag{78}$$

$$\bar{\rho}(\eta = 0) = T_w/T_\infty \Rightarrow y_4(\eta = 0) = T_w/T_\infty \tag{79}$$

$$f'(\eta \to \infty) = 1 \quad \Rightarrow y_2(\eta \to \infty) = 1 \tag{80}$$

$$\bar{\rho}(\eta \to \infty) = 1 \quad \Rightarrow y_4(\eta \to \infty) = 1. \tag{81}$$

The functions can be introduced in Julia as shown in Listing 1, where $c\mu$ is the second coefficient of the Sutherland Viscosity Law, $T\infty$ is the temperature at the boundary-layer edge, $M\infty$ is the Mach number at the boundary-layer edge, γ is the specific heat ratio, Pr is the Prandtl number and y_1, y_2, y_3, y_4, and y_5 are the terms given in Equations (64)–(66), Equation (67), and Equation (68). In the functions given in Listing 1, only 2 parameters are dimensional, which are $c\mu$ and $T\infty$. In this tutorial paper, Kelvin is the unit of both parameters. If the temperature unit is required to be different, such as Fahrenheit or Rankine, the units of $c\mu$ and $T\infty$ must be transformed into the new unit accordingly.

Listing 1. Implementation of system of equations in Julia environment. There are five functions which correspond to five first-order ordinary differential equations.

```
function Y1(y2)
    return y2
end

function Y2(y3)
    return y3
end

function Y3(y1,y3,y4,y5,cμ,T∞)
    return
−y3 ∗ ((y5/(2 ∗ (y4))) − (y5/(y4 + cμ/T∞))) − y1 ∗ y3 ∗ ((y4 + cμ/T∞)/(sqrt(y4) ∗ (1 + cμ/T∞)))
end

function Y4(y5)
    return y5
end

function Y5(y1,y3,y4,y5,cμ,T∞,M∞,Pr,γ)
    return −y5^2∗((0.5/y4) − (1/(y4 + cμ/T∞))) − Pr ∗ y1 ∗ y5/sqrt(y4)∗
    (y4 + cμ/T∞)/(1 + cμ/T∞) − (γ − 1) ∗ Pr ∗ M∞^2∗y3^2
end
```

In this paper, implementation of the Runge–Kutta method will be provided. The derivation of the Runge–Kutta method and how it calculates the function value at the next step can be checked from Reference [49]. The implementation of the Runge–Kutta method for the compressible Blasius problem can be seen in Listing 2, where N is the number of elements. It has to be emphasized that the number of node points is $N + 1$, which means that terms must be calculated until $(N + 1)^{th}$ node. The first point is the boundary condition, so there will be N number of calculations.

The initialization and the boundary conditions can be introduced as shown in Listing 3, where adi is a flag for the adiabatic or isothermal condition selection and Tw is the dimensionless wall temperature. It is nondimensionalized with T_e, so if the temperature at the boundary-layer edge, T_e, is 300 K and the wall temperature is required to be 150 K, Tw must be entered as 0.5. Another important point about Listing 3 is the indices. In the derived formulations, indices start from 0. However, in both Julia and MATLAB, indices start from 1. This is the reason why indices are starting from 1 in Listing 3 boundary conditions part.

Listing 2. Implementation of Runge-Kutta method in Julia environment. It requires four slope calculation to estimate the function value in the next node value.

```
function RK(N,Δη,y1,y2,y3,y4,y5,cμ,T∞,Pr,γ,M∞)
    for i = 1 : N
        #First Step
        k11 = Y1(y2[i])
        k21 = Y2(y3[i])
        k31 = Y3(y1[i],y3[i],y4[i],y5[i],cμ,T∞)
        k41 = Y4(y5[i])
        k51 = Y5(y1[i],y3[i],y4[i],y5[i],cμ,T∞,M∞,Pr,γ)

        #Second Step
        k12 = Y1(y2[i] + 0.5 * Δη * k21)
        k22 = Y2(y3[i] + 0.5 * Δη * k31)
        k32 = Y3(y1[i] + 0.5 * Δη * k11,y3[i] + 0.5 * Δη * k31,y4[i] + 0.5 * Δη * k41,y5[i] + 0.5 * Δη * k51,cμ,T∞)
        k42 = Y4(y5[i] + 0.5 * Δη * k51)
        k52 = Y5(y1[i] + 0.5 * Δη * k11,y3[i] + 0.5 * Δη * k31,y4[i] + 0.5 * Δη * k41,y5[i] + 0.5 * Δη * k51,cμ,T∞,M∞,Pr,γ)

        #Third Step
        k13 = Y1(y2[i] + 0.5 * Δη * k22)
        k23 = Y2(y3[i] + 0.5 * Δη * k32)
        k33 = Y3(y1[i] + 0.5 * Δη * k12,y3[i] + 0.5 * Δη * k32,y4[i] + 0.5 * Δη * k42,y5[i] + 0.5 * Δη * k52,cμ,T∞)
        k43 = Y4(y5[i] + 0.5 * Δη * k52)
        k53 = Y5(y1[i] + 0.5 * Δη * k12,y3[i] + 0.5 * Δη * k32,y4[i] + 0.5 * Δη * k42,y5[i] + 0.5 * Δη * k52,cμ,T∞,M∞,Pr,γ)

        #Fourth Step
        k14 = Y1(y2[i] + Δη * k23)
        k24 = Y2(y3[i] + Δη * k33)
        k34 = Y3(y1[i] + Δη * k13,y3[i] + Δη * k33,y4[i] + Δη * k43,y5[i] + Δη * k53,cμ,T∞)
        k44 = Y4(y5[i] + Δη * k53)
        k54 = Y5(y1[i] + Δη * k13,y3[i] + Δη * k33,y4[i] + Δη * k43,y5[i] + Δη * k53,cμ,T∞,M∞,Pr,γ)

        #Next Point Calculation
        y5[i+1] = y5[i] + (1/6) * (k51 + 2 * k52 + 2 * k53 + k54) * Δη
        y4[i+1] = y4[i] + (1/6) * (k41 + 2 * k42 + 2 * k43 + k44) * Δη
        y3[i+1] = y3[i] + (1/6) * (k31 + 2 * k32 + 2 * k33 + k34) * Δη
        y2[i+1] = y2[i] + (1/6) * (k21 + 2 * k22 + 2 * k23 + k24) * Δη
        y1[i+1] = y1[i] + (1/6) * (k11 + 2 * k12 + 2 * k13 + k14) * Δη
    end
    return y1,y2,y3,y4,y5
end
```

In the system of equations, there are five equations and five boundary conditions; however, two boundary conditions are located at the end of the domain. In order to start the calculation, all values at the $\eta = 0$ should be given. α_0 and β_0 in the Listing 3 are the initial guesses for the missing boundary conditions. They can be any value. Once they are introduced to the system, compressible Blasius equations can be solved. When the equations are solved with guessed initial conditions, the solution vector must satisfy the boundary conditions at the end of the domain. However, it will not converge at the first try because the guessed boundary conditions are not correct. To overcome this problem, different methods can be used, such as the shooting method, bisection method, or Newton's iteration method. In this paper, Newton's iteration method will be used because it is fast and it is not hard to implement. In order to use it, the algorithm needs to run with the initial guesses one time. Once the y_2 (corresponds to u) and y_4 (corresponds to T) at the end of the domain are obtained, an arbitrary small number can be added to one of the initial guesses. The algorithm can be run one more time with the new boundary condition guesses. After that, the same small number can be added to the other initial guess and the algorithm can be run one more time. After running the algorithm 3 times, there will be 3 different y_2–y_4 pairs. It has to be noted that when the small number is added to the second boundary condition (in the third run), other boundary conditions should be equal to the value in the first run. In other words, after adding a small value in the second run, it should be subtracted in the third run. The main purpose of running three times is to determine the more accurate boundary condition guess. The new boundary conditions can be calculated with:

$$\alpha = \alpha + d\alpha \tag{82}$$
$$\beta = \beta + d\beta, \tag{83}$$

where α and β are the initially guessed boundary conditions. $d\alpha$ and $d\beta$ are required for the new boundary conditions. These values can be approximated from the Taylor series expansion of the y_2 and y_4, which can be shown as:

$$y_{2,new} = y_{2,old} + \frac{\partial y_2}{\partial \alpha} d\alpha + \frac{\partial y_2}{\partial \beta} d\beta + O(d\alpha^2, d\beta^2) \tag{84}$$
$$y_{4,new} = y_{4,old} + \frac{\partial y_4}{\partial \alpha} d\alpha + \frac{\partial y_4}{\partial \beta} d\beta + O(d\alpha^2, d\beta^2). \tag{85}$$

$y_{2,new}$ and $y_{4,new}$ must be 1 due to the boundary conditions. The new system of equations for the $d\alpha$ and $d\beta$ will be:

$$\begin{bmatrix} \frac{\partial y_2}{\partial \alpha} & \frac{\partial y_2}{\partial \beta} \\ \frac{\partial y_4}{\partial \alpha} & \frac{\partial y_4}{\partial \beta} \end{bmatrix} \begin{bmatrix} d\alpha \\ d\beta \end{bmatrix} = \begin{bmatrix} 1 - y_{2,old} \\ 1 - y_{4,old} \end{bmatrix}. \tag{86}$$

The partial differentials can be approximated with the finite difference as:

$$\frac{\partial y_2}{\partial \alpha} = \frac{y_2(\alpha + \Delta) - y_2(\alpha)}{\Delta} = \frac{y_{2,new,1} - y_{2,old}}{\Delta} \tag{87}$$
$$\frac{\partial y_4}{\partial \alpha} = \frac{y_4(\alpha + \Delta) - y_4(\alpha)}{\Delta} = \frac{y_{4,new,1} - y_{4,old}}{\Delta} \tag{88}$$
$$\frac{\partial y_2}{\partial \beta} = \frac{y_2(\beta + \Delta) - y_2(\beta)}{\Delta} = \frac{y_{2,new,2} - y_{2,old}}{\Delta} \tag{89}$$
$$\frac{\partial y_4}{\partial \beta} = \frac{y_4(\beta + \Delta) - y_4(\beta)}{\Delta} = \frac{y_{4,new,2} - y_{4,old}}{\Delta}, \tag{90}$$

where $y_{2,old}$ and $y_{4,old}$ are the values obtained from the first run, $y_{2,new,1}$ and $y_{4,new,1}$ are the values obtained from the second run, and $y_{2,new,2}$ and $y_{4,new,2}$ are the values obtained from the third run. Once everything is calculated, the system of equations in Equation (86) can be used to calculate $d\alpha$ and $d\beta$. The implementation of the explained method in Julia can be seen in Listing 4.

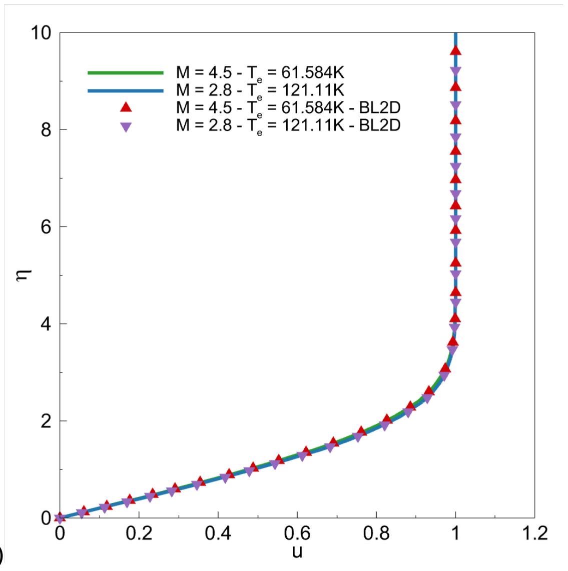
Listing 3. Initialization of the variables and implementation of boundary conditions in Julia environment. The boundary conditions for adiabatic and isothermal conditions are different than each other.

```
# Initializing the solution vectors
y1 = zeros(N+1)    # f
y2 = zeros(N+1)    # f'
y3 = zeros(N+1)    # f''
y4 = zeros(N+1)    # ρ(η)
y5 = zeros(N+1)    # ρ(η)'
η = [i*Δη for i = 0:N]
adi = 1 # adi=1 (Adiabatic) adi=0 (Isothermal)

if adi == 1
# Adibatic Boundary Conditions
    y1[1] = 0
    y2[1] = 0
    y5[1] = 0

    α = 0.1   # Initial Guess
    β = 3.0   # Initial Guess
elseif adi == 0
# Isothermal Boundary Conditions
    y1[1] = 0
    y2[1] = 0
    y4[1] = Tw # Dimensionless Wall Temperature

    α = 0.1   # Initial Guess
    β = 3.0   # Initial Guess
end
```

The same procedure will run until y_2, and y_4 at the end of the domain will be 1. It is important to decide the upper limit of the domain. If it is small, it will force the value at that point to be 1 where it should not be. It is also important to choose the small number, Δ, smaller than convergence criteria which will finalize the simulation. If Δ is higher than the convergence criteria, the simulation might run until it reaches the maximum iteration number. In the code provided in GitHub, convergence criteria is taken as 1×10^{-9} and the small number is taken as 1×10^{-10}. The results of the code for $M = 4.5$ and $M = 2.8$ are illustrated in Figure 3, where total temperatures are 311 K for both. The freestream temperature is calculated from isentropic relation and it is 61.584 K for $M = 4.5$ and 121.11 K for $M = 2.8$. The results are compared with the Iyer's [20] BL2D boundary-layer solver, which is used in NASA's well-known compressible boundary-layer stability solver LASTRAC [21].

Listing 4. Implementation of Newton's Iteration Method in Julia environment. It requires three function calls to estimate the missing boundary condition value. Each estimation will lead to closer boundary condition guess.

```
$y3[1] = \alpha$     # Initial Guess
$y4[1] = \beta$     # Initial Guess

# First solution for Newton's iteration
$y1, y2, y3, y4, y5 = RK(N, \Delta\eta, y1, y2, y3, y4, y5, c\mu, T\infty, Pr, \gamma, M\infty)$

# Storing the freestream values for Newton's iteration method
$y_{2o} = y2[N+1]$
$y_{4o} = y4[N+1]$

# Small number addition for Newton's iteration method
$y3[1] = \alpha + \Delta$   # Initial Guess + Small number
$y4[1] = \beta$     # Initial Guess

# Second solution for Newton's iteration
$y1, y2, y3, y4, y5 = RK(N, \Delta\eta, y1, y2, y3, y4, y5, c\mu, T\infty, Pr, \gamma, M\infty)$

# Storing the freestream values for Newton's iteration method
$y_{2n1} = y2[N+1]$
$y_{4n1} = y4[N+1]$

# Small number addition for Newton's iteration method
$y3[1] = \alpha$     # Initial Guess
$y4[1] = \beta + \Delta$   # Initial Guess + Small number

# Third solution for Newton's iteration
$y1, y2, y3, y4, y5 = RK(N, \Delta\eta, y1, y2, y3, y4, y5, c\mu, T\infty, Pr, \gamma, M\infty)$

# Storing the freestream values for Newton's iteration method
$y_{2n2} = y2[N+1]$
$y_{4n2} = y4[N+1]$

# Calculation of the next initial guess with Newton's iteration method
$p_{11} = (y_{2n1} - y_{2o})/\Delta$
$p_{21} = (y_{4n1} - y_{4o})/\Delta$
$p_{12} = (y_{2n2} - y_{2o})/\Delta$
$p_{22} = (y_{4n2} - y_{4o})/\Delta$
$r_1 = 1 - y_{2o}$
$r_2 = 1 - y_{4o}$
$\Delta\alpha = (p_{22} * r_1 - p_{12} * r_2)/(p_{11} * p_{22} - p_{12} * p_{21})$
$\Delta\beta = (p_{11} * r_2 - p_{21} * r_1)/(p_{11} * p_{22} - p_{12} * p_{21})$
$\alpha = \alpha + \Delta\alpha$
$\beta = \beta + \Delta\beta$
```



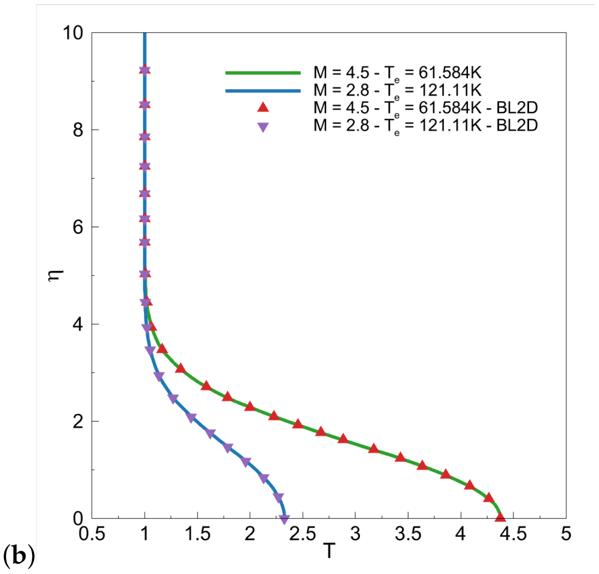


Figure 3. The distribution of the (**a**) velocity and (**b**) temperature of the compressible Blasius equation obtained by the given code and BL2D boundary-layer solver [20] for freestream Mach number 2.8 and 4.5 where freestream temperatures are 121.11 K and 61.584 K, respectively.

3. Comparison of Julia and MATLAB

The design process requires lots of simulations in order to obtain the final and optimized design. It is highly beneficial to have a fast CFD solver. One of the crucial factors that affects the speed of the solver is the language. The same script may lead to different central processing unit (CPU) times with different coding languages. Additionally, similar simulations will be required multiple times. Eventually, the total time spent on simulations might be drastic with a slow solver.

MATLAB is one of the languages that is widely used. It is one of the favorite coding language for most of the students because of its user-friendly syntax, easy debugging feature, and built-in functions. One of the most important drawbacks of this language is that it is not free. It is also slower than high-performance languages, such as Fortran and C/C++. Julia is a user-friendly, open-source language that can increase productivity drastically [13]. Another great feature of Julia is that it is completely free. Julia can call C, Fortran, and Python libraries. It is great for experienced engineers who think that their previous code in other coding languages will be useless.

One of the great concerns about language selection is the speed of the code. For large-scaled projects, most of the time, a fast solver is the most important point. In this section, the same problem will be solved with Julia and MATLAB codes. The solution times will be compared with each other. The purpose of this comparison is to provide a rough estimation about code execution speeds. Three different test cases will be applied for both languages. The cases are unsteady inviscid Burgers' equation with the first-order backward finite difference scheme, heat equation with second-order central finite difference scheme, and compressible Blasius equation with fourth-order Runge–Kutta and Newton's iteration method. Burgers' and heat equations will be tested with for loops with file operations, vectorized operations with file operations, for loops without file operations, and vectorized operations without file operations. For the compressible Blasius equation solver, the code developed for this paper will be used. The test cases will simulate real-life problems by solving the problem and exporting the solution vector to a text file when the file operations are included. In real-life problems, most of the time, post-processing is required after the simulation. In order to do that, saving data into a file is required. If it is a steady problem, exporting can be done at the end of the simulation, if there are not any other limitations or additional requirements. On the other hand, if the problem is unsteady, exporting the data in different time steps during the simulations is required. This is the reason why there will be two different simulations where data will be exported and will not be exported.

The first case is unsteady, inviscid, Burgers' equation in one dimension, which can be represented in the conservative form:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right). \tag{91}$$

The equation is solved with a first-order backward finite difference scheme. The details of the scheme will not be provided because the purpose of the test case is to measure the speed difference of two similarly developed codes. However, codes that are used in this paper are available on GitHub. Interested readers can check the implementation details from the codes. The number of elements in the problem is taken as 2500, 5000, and 10,000. The same simulation will be run with an increasing number of elements to show the solution time change trend. The execution time will be calculated by *BenchmarkTools* in Julia and *tic/toc* functions in MATLAB. The standard deviation will be calculated manually by using 10 data points obtained from the runs. The time step is taken as half of the grid spacing. The domain is limited with [0, π] and the initial conditions for velocity, $u(x_i, t)$, are taken as:

$$u(x_i, 0) = sin(x_i). \tag{92}$$

The solution vector is written to a "$.txt$" file for every hundredth iteration. The mean execution times with the standard deviation of the data obtained by Julia and MATLAB solvers are given in Tables 1 and 2. Table 1 provides the execution times with file operations, and Table 2 excludes file operations in the calculations. The results show that MATLAB is slow with file operations. There is approximately 15 times' difference between Julia and MATLAB mean execution times with file operations and for loops, but the speed-up difference is decreasing to 8 with vectorization. Without file operations, Julia is 3 times faster than MATLAB with for loops. However, MATLAB vectorization is faster than Julia. One interesting point of this test case is that MATLAB execution times are reduced by vectorization, except for the N = 10,000 case with file operations excluded. Detailed investigations about this trend indicate that there is a relation between the L1, L2, L3 cache size of the CPU and the vectorization performance. Tests are completed in 3 different computers with varying cache sizes. After a certain number of elements, vectorization starts to increase the mean execution time. The limiting number of elements is related to cache size. In computers with higher cache sizes, the negative effect of vectorization started after N = 5000. In computers with lower cache sizes, the negative effect started after N = 2500. This trend is not observed in Julia language. In Julia, for loops are highly optimized and manual vectorization leads to an increase in the mean execution times because manual vectorization creates temporary arrays during the calculations. Creating and deleting these temporary arrays require more time than calculation with for loops. In MATLAB, results indicate that temporary array usage is faster than for loops up to certain array size.

Table 1. Mean execution times and standard deviations of the Burgers' equation solver written in MATLAB and Julia by including file operations. The mean execution times are given in second. Ten data points are used in the calculation of the mean and the standard deviation.

File Op. Included		**N = 2500**		**N = 5000**		**N = 10,000**	
		Julia	**MATLAB**	**Julia**	**MATLAB**	**Julia**	**MATLAB**
for Loop	Mean	0.0360	0.5214	0.1394	1.9817	0.5592	7.8060
	STD	0.0010	0.0137	0.0017	0.0210	0.0062	0.0522
Vectorized	Mean	0.0643	0.5145	0.2678	1.9471	1.0042	7.7527
	STD	0.0139	0.0137	0.0287	0.0107	0.0348	0.0402

Table 2. Mean execution times and standard deviations of the Burgers' equation solver written in MATLAB and Julia by excluding file operations. The mean execution times are given in second. Ten data points are used in the calculation of the mean and the standard deviation.

File Op. Excluded		**$N = 2500$**		**$N = 5000$**		**$N = 10{,}000$**	
		Julia	**MATLAB**	**Julia**	**MATLAB**	**Julia**	**MATLAB**
for Loop	Mean	0.0059	0.0177	0.0233	0.0651	0.0930	0.2562
	STD	0.0001	0.0031	0.0011	0.0024	0.0005	0.0138
Vectorized	Mean	0.0437	0.0166	0.0866	0.0564	0.4542	0.3091
	STD	0.0117	0.0021	0.0215	0.0040	0.0584	0.0138

In the previous test case, the one-dimensional Burgers' equation is solved. For the second test case, the two-dimensional heat equation is solved. The two-dimensional heat equation can be shown as:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right), \tag{93}$$

where α is a constant which is taken as $0.25\Delta x$. The time step is taken as Δx. This assures that the coefficient of the second derivative will satisfy the stability condition. The boundary conditions of the system are 1 for each side and the initial conditions for the remaining nodes are 0. The heat equation is solved with the second-order central finite difference with 250×250, 500×500, and 1000×1000 elements. The domain is limited with $[0, \pi]^2$. The execution times of the two codes are given in Table 3 with file operations and in Table 4 without file operations. For this problem, the results indicate that Julia file operations are faster as it is observed in Burgers' equation solver. Vectorization has a negative effect for all cases in this problem. For Julia, vectorization increases solution time approximately 8 times without file operations, and 4 times with file operations. On the other hand, MATLAB vectorization increases the solution time approximately twice without file operations and 1.2 times with file operations. Julia with *for* loops has the fastest solution time for all cases. It is approximately 2.5 times faster than MATLAB without file operations and approximately 8 times faster with file operations.

Table 3. Mean execution times and standard deviations of the heat equation solver written in MATLAB and Julia by including file operations. The mean execution times are given in second. Ten data points are used in the calculation of the mean and the standard deviation.

File Op. Included		**$N = 250 \times 250$**		**$N = 500 \times 500$**		**$N = 1000 \times 1000$**	
		Julia	**MATLAB**	**Julia**	**MATLAB**	**Julia**	**MATLAB**
for Loop	Mean	0.4604	3.1921	3.3852	22.8472	25.1985	151.5271
	STD	0.0160	0.4149	0.0132	0.4356	0.0470	0.5615
Vectorized	Mean	1.4524	3.2964	10.0283	28.4063	81.2305	188.5569
	STD	0.1622	0.1531	0.4356	1.0269	0.4925	0.9425

Table 4. Mean execution times and standard deviations of the heat equation solver written in MATLAB and Julia by excluding file operations. The mean execution times are given in second. Ten data points are used in the calculation of the mean and the standard deviation.

File Op. Excluded		**$N = 250 \times 250$**		**$N = 500 \times 500$**		**$N = 1000 \times 1000$**	
		Julia	**MATLAB**	**Julia**	**MATLAB**	**Julia**	**MATLAB**
for Loop	Mean	0.1538	0.3268	1.1964	3.7574	10.5993	28.5775
	STD	0.0038	0.0187	0.0077	0.3274	0.1667	0.1065
Vectorized	Mean	0.8056	0.5087	8.5873	9.3634	71.9560	67.5810
	STD	0.0706	0.0257	0.0687	0.9880	0.4309	0.8318

Lastly, the derived compressible Blasius equations for the present study will be solved in both MATLAB and Julia. The difference of that case is to test the function calls because sometimes dividing the solver into smaller functions may lead to longer solution times. The problem will be solved with 50,000, 100,000, and 200,000 elements. Table 5 gives the solution times of two codes developed in MATLAB and Julia. In this problem, Julia is drastically faster than MATLAB, and the time differences are increasing with the problem size. With 50,000 elements, Julia is approximately 15 times faster than MATLAB, with 100,000 elements, it is 32 times faster, and with 200,000 elements, it is 120 times faster.

Table 5. Mean execution times and standard deviations of the compressible Blasius equations solver written in MATLAB and Julia. The mean execution times are given in second. Ten data points are used in the calculation of the mean and the standard deviation.

File Op. Excluded		$N = $ **50,000**		$N = $ **100,000**		$N = $ **200,000**	
		Julia	**MATLAB**	**Julia**	**MATLAB**	**Julia**	**MATLAB**
for Loop	Mean	0.0831	1.2468	0.1631	5.1070	0.3298	39.4378
	STD	0.0054	0.0310	0.0057	0.3006	0.0098	0.8308

Although time differences are varying with problems, Julia with for loops exhibited better performance than MATLAB in every problem. On the other hand, MATLAB showed better performance when both of the codes are developed in vectorized form. In general, MATLAB file operations are slower than Julia. It has to be noted that MATLAB has special data exporting options which might be faster, such as $.mat$ extensions. In order to conduct an exact comparison, regular $.txt$ extension with conventional exporting commands are used. The main purpose of these time comparisons is to provide an approximate performance differences between Julia and MATLAB under different conditions. In this paper, Julia is compared with MATLAB. Interested readers can check Lubin and Dunning's paper [50] for other coding language comparisons.

4. Conclusions

Compressible Blasius equation, which comes from boundary-layer theory, is extensively used by researchers to validate the CFD simulation results. One can estimate the number of elements required to capture the boundary-layer by using the solution of the compressible Blasius equation. Although it is crucial to understand the boundary-layer theory, undergraduate- or graduate-level boundary-layer classes may not be adequate for a student to fully understand it due to time limitations. A step-by-step tutorial may help students to understand the theory better. Both compressible and incompressible boundary-layer problems require numerical solution. Deriving the equations from scratch and implementing the numerical methods may shorten the learning curve for a student or an engineer.

In this paper, compressible Blasius equation and energy equation are derived from scratch. The final system of equations is solved in the Julia environment. For the numerical implementation, the fourth-order Runge–Kutta and Newton's iteration methods are employed. It has to be noted that other methods such as Runge–Kutta–Fehlberg, compact finite difference, a high-order finite-difference can also be used to solve the final system of equations. However, authors preferred the Runge–Kutta and Newton's iteration method due to their accuracy and wide usage in the literature. Moreover, the authors compared the Julia and MATLAB solver speed to give an initial impression about performance of Julia. The results showed that MATLAB is slower than Julia in file operations. Additionally, Julia is faster than MATLAB with for loops. On the other hand, Julia vectorization affects the solution times negatively. However, MATLAB vectorization decreases the solution time for small-sized problems. When the problem size increases, MATLAB vectorization also has a negative effect on the solution time. It has to be noted that these test cases are relatively less demanding cases. In real-life problems, simulations require longer codes with more complex operations. In longer runs, the time difference in between these two languages may increase.

Author Contributions: Code generation, F.O. and K.K.; validation, F.O. and K.K.; writing—original draft preparation, F.O.; writing—review and editing, F.O. and K.K.; visualization, F.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All the data used and generated in this study are available in the GitHub link provided in Appendix A.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Julia setup files can be downloaded from their website (https://julialang.org/downloads/ (accessed on 4 November 2021)). The website also includes instructions on how to install Julia on Windows, Linux, and MAC operating systems. Some of the useful resources for learning Julia are listed below:

- https://docs.julialang.org/en/v1/ (accessed on 4 November 2021)
- https://www.coursera.org/learn/julia-programming (accessed on 4 November 2021)
- https://www.youtube.com/user/JuliaLanguage/featured (accessed on 4 November 2021)
- https://www.youtube.com/user/Parallel Computing and Scientific Machine Learning (accessed on 4 November 2021)
- https://discourse.julialang.org/ (accessed on 4 November 2021)

It is common to use external packages for Julia. In order to do that, Pkg, which is Julia's built-in package manager, can be used. Once Julia is opened, Pkg can be activated with the "]" button in Windows. In Linux, calling "julia" in the terminal will open it. After that, "Pkg.add("Pluto")" will trigger the setup process for that package. In here, we used Pluto as an example because, in GitHub, our codes are developed in the Pluto environment. After Pluto is installed, Pluto can be run with "Pluto.run()". This command will open a new tab in the browser which you can run your Julia codes. After that, the "using Pluto" line must be placed to the top of the file. For "Plots" package, the commands will be "Pkg.add("Plots")" and "using Plots". Since the Plots package does not have a GUI, there is not a command called "Plots.run()".

Other than Pluto, JuliaPro, which includes Julia and the Juno IDE (https://juliacomputing.com/products/juliapro/ (accessed on 4 November 2021)), can be used as an editor and compiler. This software contains a set of packages for plotting, optimization, machine learning, database, and much more. Pluto is appropriate for small scripts, while JuliaPro is better for more complex codes. The GitHub link of the codes used in this paper is:

- https://github.com/frkanz/A-CFD-Tutorial-in-Julia-Compressible-Blasius/tree/main (accessed on 4 November 2021)

References

1. Anderson, J.D. *Fundamentals of Aerodynamics*; McGraw-Hill Education: New York, NY, USA, 2010.
2. Schlichting, H.; Gersten, K. *Boundary-Layer Theory*; Springer: Berlin/Heidelberg, Germany, 2016.
3. Anderson, J.D. Ludwig Prandtl's Boundary Layer. *Phys. Today* **2005**, *58*, 42–48. [CrossRef]
4. Prandtl, L. Über Flüssigkeitsbewegung bei sehr kleiner Reibung, Verh 3 int. Math-Kongr, Heidelberg, English Translation. 1904. Availabel online: http://homepage.ntu.edu.tw/~wttsai/Adv_Fluid/NACA_TM-452.pdf (accessed on 4 November 2021).
5. Blasius, H. Grenzschichten in Flüssigkeiten mit Kleiner Reibung. *Z. Math. Phys.* **1908**, *60*, 397–398.
6. Hager, W.H. Blasius: A life in research and education. *Exp. Fluids* **2003**, *34*, 566–571. [CrossRef]
7. Cousteix, T.; Cebeci, J. *Modeling and Computation of Boundary-Layer Flows*; Springer: Berlin/Heidelberg, Germany, 2005.
8. White, F.M.; Corfield, I. *Viscous Fluid Flow*; McGraw-Hill: New York, NY, USA, 2006; Volume 3.
9. Metcalf, M.; Reid, J.K. *Fortran 90/95 Explained*; Oxford University Press, Inc.: Oxford, UK, 1999.

10. Sanner, M.F. Python: A programming language for software integration and development. *J. Mol. Graph. Model.* **1999**, *17*, 57–61. [PubMed]
11. Stroustrup, B. *The C++ Programming Language*; Pearson Education: London, UK, 2000.
12. MATLAB. *Version 7.10. 0 (R2010a)*; The MathWorks Inc.: Natick, MA, USA, 2010.
13. Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V.B. Julia: A fresh approach to numerical computing. *SIAM Rev.* **2017**, *59*, 65–98. [CrossRef]
14. Barba, L.; Forsyth, G. CFD Python: the 12 steps to Navier-Stokes equations. *J. Open Source Educ.* **2018**, *2*, 21. [CrossRef]
15. Oliphant, T.E. *A Guide to NumPy*; Trelgol Publishing USA: Natick, MA, USA, 2006; Volume 1.
16. Ketcheson, D.I. Teaching numerical methods with IPython notebooks and inquiry-based learning. In Proceedings of the 13th Python in Science Conference, Austin, TX, USA, 6–12 July 2014; pp. 19–24.
17. Ketcheson, D.I.; Mandli, K.; Ahmadia, A.J.; Alghamdi, A.; de Luna, M.Q.; Parsani, M.; Knepley, M.G.; Emmett, M. PyClaw: Accessible, extensible, scalable tools for wave propagation problems. *SIAM J. Sci. Comput.* **2012**, *34*, 210–231. [CrossRef]
18. Pawar, S.; San, O. CFD Julia: A learning module structuring an introductory course on computational fluid dynamics. *Fluids* **2019**, *4*, 159. [CrossRef]
19. Oz, F.; Kara, K. A CFD Tutorial in Julia: Introduction to Laminar Boundary-Layer Theory. *Fluids* **2021**, *6*, 207. [CrossRef]
20. Iyer, V. *Computer Program BL2D for Solving Two-Dimensional and Axisymmetric Boundary Layers*; NASA NASA-CR-4668; NASA: Washington, DC, USA, 1995.
21. Chang, C.L. *Langley Stability and Transition Analysis Code (LASTRAC) Version 1.2 User Manual*; NASA TM-2004-213233; NASA: Washington, DC, USA, June 2004.
22. Brennan, G.; Gajjar, J.; Hewitt, R. Tollmien–Schlichting wave cancellation via localised heating elements in boundary layers. *J. Fluid Mech.* **2021**, *909*. [CrossRef]
23. Brennan, G.S.; Gajjar, J.S.; Hewitt, R.E. Cancellation of Tollmien–Schlichting waves with surface heating. *J. Eng. Math.* **2021**, *128*, 1–23. [CrossRef]
24. Corelli Grappadelli, M.; Sattler, S.; Scholz, P.; Radespiel, R.; Badrya, C. Experimental investigations of boundary layer transition on a flat plate with suction. In Proceedings of the AIAA Scitech 2021 Forum, Virtual Event, 11–15 and 19–21 January 2021; p. 1452.
25. Rigas, G.; Sipp, D.; Colonius, T. Nonlinear input/output analysis: Application to boundary layer transition. *J. Fluid Mech.* **2021**, *911*. [CrossRef]
26. Haley, C.; Zhong, X. Supersonic mode in a low-enthalpy hypersonic flow over a cone and wave packet interference. *Phys. Fluids* **2021**, *33*, 054104. [CrossRef]
27. Malik, M.R. Numerical methods for hypersonic boundary layer stability. *J. Comput. Phys.* **1990**, *86*, 376–413. [CrossRef]
28. Fedorov, A. Transition and stability of high-speed boundary layers. *Annu. Rev. Fluid Mech.* **2011**, *43*, 79–95. [CrossRef]
29. Long, T.; Dong, Y.; Zhao, R.; Wen, C. Mechanism of stabilization of porous coatings on unstable supersonic mode in hypersonic boundary layers. *Phys. Fluids* **2021**, *33*, 054105. [CrossRef]
30. Fong, K.D.; Wang, X.; Zhong, X. Numerical simulation of roughness effect on the stability of a hypersonic boundary layer. *Comput. Fluids* **2014**, *96*, 350–367. [CrossRef]
31. Kara, K.; Balakumar, P.; Kandil, O. Receptivity of hypersonic boundary layers due to acoustic disturbances over blunt cone. In Proceedings of the 45th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, 8–11 January 2007; p. 945.
32. Kara, K.; Balakumar, P.; Kandil, O. Effects of wall cooling on hypersonic boundary layer receptivity over a cone. In Proceedings of the 38th Fluid Dynamics Conference and Exhibit, Seattle, WA, USA, 23–26 June 2008; p. 3734.
33. Kara, K.; Balakumar, P.; Kandil, O.A. Effects of nose bluntness on hypersonic boundary-layer receptivity and stability over cones. *AIAA J.* **2011**, *49*, 2593–2606. [CrossRef]
34. Oz, F.; Kara, K. Effects of Local Cooling on Hypersonic Boundary-Layer Stability. In *AIAA Scitech 2021 Forum*; AIAA: Reston, VA, USA, 2021; p. 0940.
35. Drozdz, A.; Niegodajew, P.; Romanczyk, M.; Sokolenko, V.; Elsner, W. Effective use of the streamwise waviness in the control of turbulent separation. *Exp. Therm. Fluid Sci.* **2021**, *121*. [CrossRef]
36. Iyer, P.S.; Malik, M.R. Wall-modeled LES of flow over a Gaussian bump. In *AIAA Scitech 2021 Forum*; AIAA: Reston, VA, USA, 2021; p. 1438.
37. Mohammed-Taifour, A.; Weiss, J. Periodic forcing of a large turbulent separation bubble. *J. Fluid Mech.* **2021**, *915*. [CrossRef]
38. Hady, F.; Ibrahim, F.; Abdel-Gaied, S.; Eid, M. Effect of heat generation/absorption on natural convective boundary-layer flow from a vertical cone embedded in a porous medium filled with a non-Newtonian nanofluid. *Int. Commun. Heat Mass Transf.* **2011**, *38*, 1414–1420. [CrossRef]
39. Hady, F.M.; Ibrahim, F.S.; Abdel-Gaied, S.M.; Eid, M.R. Radiation effect on viscous flow of a nanofluid and heat transfer over a nonlinearly stretching sheet. *Nanoscale Res. Lett.* **2012**, *7*, 1–13. [CrossRef]
40. Hady, F.; Ibrahim, F.; Abdel-Gaied, S.; Eid, M.R. Boundary-layer non-Newtonian flow over vertical plate in porous medium saturated with nanofluid. *Appl. Math. Mech.* **2011**, *32*, 1577–1586. [CrossRef]
41. Hady, F.; Ibrahim, F.; Abdel-Gaied, S.; Eid, M. Boundary-layer flow in a porous medium of a nanofluid past a vertical cone. In *An Overview of Heat Transfer Phenomena*; Kazi, S.N., Ed.; IntechOpen: London, UK, 2012; pp. 91–104.
42. Sohail, M.; Naz, R.; Abdelsalam, S.I. Application of non-Fourier double diffusions theories to the boundary-layer flow of a yield stress exhibiting fluid model. *Phys. Stat. Mech. Appl.* **2020**, *537*, 122753. [CrossRef]

43. Bhatti, M.; Alamri, S.Z.; Ellahi, R.; Abdelsalam, S.I. Intra-uterine particle–fluid motion through a compliant asymmetric tapered channel with heat transfer. *J. Therm. Anal. Calorim.* **2020**, *144*, 2259–2267. [CrossRef]
44. Tannehill, J.C.; Pletcher, R.H.; Anderson, D.A. *Computational Fluid Mechanics and Heat Transfer*; Taylor & Francis: Bristol, PA, USA, 1997.
45. National Center for Biotechnology Information. PubChem Periodic Table of Elements. 2021. Available online: https://pubchem.ncbi.nlm.nih.gov/element/Titanium (accessed on 12 October 2021).
46. Howarth, L. Concerning the effect of compressibility on lam inar boundary layers and their separation. *Proc. R. Soc. London. Ser. Math. Phys. Sci.* **1948**, *194*, 16–42.
47. LII, W.S. The viscosity of gases and molecular force. *Lond. Edinb. Dublin Philos. Mag. J. Sci.* **1893**, *36*, 507–531.
48. Moin, P. *Fundamentals of Engineering Numerical Analysis*; Cambridge University Press: Cambridge, UK, 2010.
49. Anderson, J.D.; Degrez, G.; Dick, E.; Grundmann, R. *Computational Fluid Dynamics: An Introduction*; Springer Science & Business Media: Berlin, Germany, 2013.
50. Lubin, M.; Dunning, I. Computing in operations research using Julia. *INFORMS J. Comput.* **2015**, *27*, 238–248. [CrossRef]