Development of an explicit pressure-based unstructured solver for three-dimensional incompressible flows with graphics hardware acceleration

Jordan Thistle
Iowa State University

Development of an explicit pressure-based unstructured solver for three-dimensional incompressible flows with graphics hardware acceleration

by

Jordan Thistle

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Aerospace Engineering

Program of Study Committee:

R. Ganesh Rajagopalan, Major Professor

Zhi J. Wang

Vinay Dayal

Iowa State University

Ames, Iowa

2012

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and gratitude to my major professor, Dr. R. Ganesh Rajagopalan, for his support and guidance throughout this research. I would also like to thank my committee members: Dr. Z.J. Wang and Dr. Vinay Dayal for their support. I greatly appreciate the financial support provided by the NSF and the IAWIND program for this research. Many thanks are due to my friends and colleagues, Daniel Garrick, Kanchan Guntupalli and William Polzin for their valuable suggestions and help. I would also convey my appreciation to the Department of Aerospace Engineering at Iowa State University and its staff for the administrative help and computing resources. I would like to express my deepest gratitude to my wife Jessica, my parents and family for their moral support, confidence and love. Finally, I would like dedicate this thesis to my mother.

CHAPTER 1. INTRODUCTION

1.1 Background

Technological advancements are the driving force of the field of computational fluid dynamics (CFD). Enhancements in the computing power of computer hardware allows for increasingly accurate methods to be used. This allows modern CFD to tackle larger and more complex flows. Traditional experiments often require great time commitments and monetary support. Implementing CFD reduces the time and financial backing necessary to complete experiments. Many engineering problems are constrained by time; any additional time advantages are highly valued. In recent years, the computation speed of CFD has been increased through the use of parallel computing. The effectiveness of parallel computing is diminished by implicit and semi-implicit solvers that require expensive serial matrix inversion processes. Additionally, in many cases the cost associated with building a parallel computing cluster is prohibitively expensive. With recent advancements in general purpose graphics processing unit computing (GPGPU), high performance parallel processing has become readily available at low cost. The parallel computing performance of GPUs have outpaced traditional CPU parallel computing in terms of scalability and speed vs. cost. In order to take advantage of this performance, the development of new algorithms is essential and programming techniques need to be modified to properly utilize the GPU hardware's architecture.

1.1.1 Time Integration Methods

There are three categories of time integration used in CFD: explicit, implicit and semi-implicit. One of the earliest method used is the first-order forward Euler method, an explicit method. The main problem with forward Euler integration is the small time step required to maintain the stability. This was followed up by the first implicit time integration, backward Euler. Since backward Euler is unconditionally stable, it eliminates the restriction on time step size, however it is only first order accurate in time. These were followed by a series of higher order implicit and semi-implicit (a combination of implicit and explicit) methods. Semi-implicit methods such as MacCormack's predictor-corrector and Crank-Nicolson allow higher accuracy in time with a reduced restriction on time step size.

One alternative for higher order time accuracy is to use the Runge-Kutta family of methods. These methods are a collection of explicit and implicit time integration methods. The explicit Runge-Kutta scheme was developed around 1900 by Runge and Kutta. This family of methods allows for more efficient programs through the reduction of expensive computations. Despite the efficiency of Runge-Kutta methods, the application of this integration scheme to incompressible flow solver such as SIMPLER has been uncommon. Researchers at Iowa State University have been developing an explicit algorithm for low speed flows. This algorithm has been shown to work in 2D structured (Purohit [1]) and unstructured (Lestari [2]). In these works, the Runge-Kutta algorithm is used to update the velocity field. This research extends this concept to take advantage of 3-dimensional tetrahedral unstructured grids.

1.1.2 GPGPU Computing

Since the introduction of discrete graphics processing units (GPUs), GPUs have rapidly increased in performance. This is largely due to the massive growth of the video game industry and the demand for increasingly stunning visual effects. The advances in GPU technology has outpaced CPU development, primarily due to the highly parallel and scalable hardware design. These chips, originally designed for graphics rendering, are built to operate under a Single Instruction Multiple Thread (SIMT) model.

With the advent of programming languages capable of harnessing the power of the GPU, such as NVIDIA's *CUDA* and ATI's *Stream*, GPGPU computing has begun to replace traditional high performance computing. This is in partly due to the simplicity of the instruction units and large floating point performance (FLOPS) for very little cost. The latest graphics card released by NVIDIA, the GTX 690, has 3072 cores at a speed of 1.1 GHz. Although the per core speed is marginally slower than the previous generation, this compensated for by increase the core count from 512 cores. By comparison, the latest Intel processor, 'Sandy Bridge', has 4 cores at a speed of 3.5 GHz. However, making high performance GPU codes requires a significant departure in programming technique and design. This research, takes advantage of a new explicit algorithm to leverage the full GPU computing power in order to reduce the computation time required by more than an order of magnitude.

1.2 Current Work

The explicit algorithm is based on the generic four stage Runge-Kutta scheme and the SIMPLER algorithm by Patankar [3]. The velocity field is updated using the Runge-Kutta four stage algorithm, while, the pressure field is obtained by solving the discretized continuity equation. The pressure field solution is similar to the precedure used in the SIMPLER algorithm. The algorithm is first developed using the general Crank-Nicolson and Fully Implicit time integration schemes. The expicit four stage Runge-Kutta integraion scheme is derived and implemented using both the CPU and CUDA. Finally, the unsteady rotor model [4] is incorporated to test the ability of the Runge-Kutta scheme to handle complex unsteady rotor flows.

CHAPTER 2. 3D UNSTRUCTURED FLOW SOLVER : THEORETICAL FORMULATION

2.1 Governing Equations

2.1.1 Conservation of Mass

The mass conservation equation, also known as the continuity equation for a general fluid flow, can be represented as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{V}) = 0 \tag{2.1}$$

Assuming an incompressible fluid, ρ is constant, the above equation reduces to:

$$\rho \nabla \cdot (\vec{V}) = 0 \tag{2.2}$$

The above equation can be expanded in 3-D coordinate system as:

$$\frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho w)}{\partial z} = 0 \tag{2.3}$$

where, in the x, y and z directions u, v and w are the velocity components, respectively.

2.1.2 Conservation of Momentum

By applying Newton's second law to an infinitesimal fluid control volume, the momentum conservation equation can be derived. The equation can be written in divergence form as:

$$\frac{\partial(\rho \vec{V})}{\partial t} + \nabla \cdot (\rho \vec{V} \vec{V}) = \rho \vec{f} + \nabla \cdot \pi_{ij} \tag{2.4}$$

in the above equation, the divergence of the stress tenser π_{ij} is given by:

$$\nabla \cdot \pi_{ij} = -(\tilde{I} \cdot \nabla)p - \nabla(\nabla \cdot \tilde{I}) + \nabla \cdot \tilde{\tau} \tag{2.5}$$

where, the term $\nabla(\nabla \cdot \tilde{I})$ goes to zero for orthogonal systems. Also, the pressure source term, $(\tilde{I} \cdot \nabla)p$ reduces to ∇p. The momentum equation thus reduces to:

$$\frac{\partial(\rho\vec{V})}{\partial t} + \nabla \cdot (\rho\vec{V}\vec{V}) = -\nabla p + \nabla \cdot \tilde{\tau} \tag{2.6}$$

Assuming a Newtonian fluid, the shear stress on a particular fluid element is linearly proportional to the deformation rate. For an isotropic and Newtonian fluid, the viscous stress tensor $\tilde{\tau}$ is given by:

$$\tilde{\tau} = \mu\left[\nabla\vec{V} + (\nabla\vec{V})^T - \frac{2}{3}(\nabla \cdot \vec{V})\tilde{I}\right] \tag{2.7}$$

For an incompressible fluid, $(\nabla \cdot \vec{V})$ and $\nabla \cdot (\nabla\vec{V})^T$ are zero. Equation 2.4 reduces to:

$$\frac{\partial(\rho\vec{V})}{\partial t} + \nabla \cdot (\rho\vec{V}\vec{V}) = -\nabla p + \nabla \cdot (\mu\nabla\vec{V}) \tag{2.8}$$

Expanding Equation 2.8 in a 3-D coordinate system and assuming viscosity is constant, the momentum equations in the x, y and z directions are:

$$\frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho uu)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} + \frac{\partial(\rho uw)}{\partial z} = -\frac{\partial p}{\partial x} + \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + S'_x \tag{2.9}$$

$$\frac{\partial(\rho v)}{\partial t} + \frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho vv)}{\partial y} + \frac{\partial(\rho vw)}{\partial z} = -\frac{\partial p}{\partial y} + \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) + S'_y \tag{2.10}$$

$$\frac{\partial(\rho w)}{\partial t} + \frac{\partial(\rho uw)}{\partial x} + \frac{\partial(\rho vw)}{\partial y} + \frac{\partial(\rho ww)}{\partial z} = -\frac{\partial p}{\partial z} + \mu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) + S'_z \tag{2.11}$$

where the S'_x, S'_y and S'_z are the momentum sources in the x, y and z directions, respectively. By combining the convective and diffusive fluxes, we can define a total flux J:

$$J_x^u = \rho uu - \mu\frac{\partial u}{\partial x} \qquad J_x^v = \rho uv - \mu\frac{\partial v}{\partial x} \qquad J_x^w = \rho uw - \mu\frac{\partial w}{\partial x} \tag{2.12}$$

$$J_y^u = \rho vu - \mu\frac{\partial u}{\partial y} \qquad J_y^v = \rho vv - \mu\frac{\partial v}{\partial y} \qquad J_y^w = \rho vw - \mu\frac{\partial w}{\partial y} \tag{2.13}$$

$$J_z^u = \rho wu - \mu\frac{\partial u}{\partial z} \qquad J_z^v = \rho wv - \mu\frac{\partial v}{\partial z} \qquad J_z^w = \rho ww - \mu\frac{\partial w}{\partial z} \tag{2.14}$$

Using the total flux J, the momentum Equations 2.9 - 2.11 can be simplified to:

$$\frac{\partial(\rho u)}{\partial t} + \frac{\partial J_z^u}{\partial z} + \frac{\partial J_y^u}{\partial y} + \frac{\partial J_z^u}{\partial z} = S_x' - \frac{\partial p}{\partial x} \tag{2.15}$$

$$\frac{\partial(\rho v)}{\partial t} + \frac{\partial J_x^v}{\partial x} + \frac{\partial J_y^v}{\partial y} + \frac{\partial J_z^v}{\partial z} = S_y' - \frac{\partial p}{\partial y} \tag{2.16}$$

$$\frac{\partial(\rho w)}{\partial t} + \frac{\partial J_x^w}{\partial x} + \frac{\partial J_y^w}{\partial y} + \frac{\partial J_z^w}{\partial z} = S_z' - \frac{\partial p}{\partial z} \tag{2.17}$$

2.2 Spatial Discretization

For a finite volume method, the computational domain is subdivided into control volumes. For each control volume the conservation laws are satisfied. When summed up, the control volumes should cover the entire physical domain without overlap and the overall flux should be conserved. For this research, the domain is subdivided into tetrahedrons and vertex centered control volumes are generated. In order to generate non-overlapping control volumes, median dual based volumes are created around each node by joining the center of the tetrahedral element to the centers of its faces. This results in a control volume with triangular control volume faces. The discretization scheme is derived such that the fluxes of the flow variables are conserved. The basic idea of vertex centered discretization for the momentum conservation equations is adopted from Baliga and Prakash [5, 6]. A typical tetrahedral element with node points (P, Q, R, S) is shown in Figures 2.1, 2.2 and 2.3 illustrate the median dual based control volumes around a representative node P.

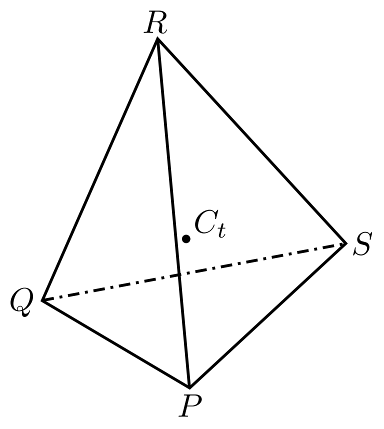


Figure 2.1 A typical tetrahedral element with its centroid at C_t

2.3 Variable Interpolation Function and Flux Calculation

In order to approximate the flow variable and the corresponding gradients within a tetrahedral element, interpolation functions need to be formulated. The following assumptions are made about the flow variables and their distribution:

- The density (ρ) and viscosity (μ) are constant over an element. This ensures the continuity of flux of any general variable (Φ) at the control volume faces.
- The source term in the momentum equations (S'_x, S'_y, S'_z) are constant over an element.
- The pressure p varies linearly over an element. This allows the pressure gradient terms in the momentum equations to be constant over an element.

Interpolation functions are needed so that fluxes can be computed through the control volume faces. The choice of interpolation function is important since, it needs to correctly model the flow physics and produce a reasonably accurate numerical solution. For example, the exponential function is the exact solution for a 1-D convection diffusion problem without source terms. However, the exponential function is very computationally expensive. As a result, the power law scheme is used to approximate the exponential function and reduce computational cost. The implementation of the interpolation function is simple in a cartesian grid but, it is complicated by the use of unstructured grids, since the control volume faces are not generally aligned with the flow direction. This may reduce the accuracy of the algorithm,

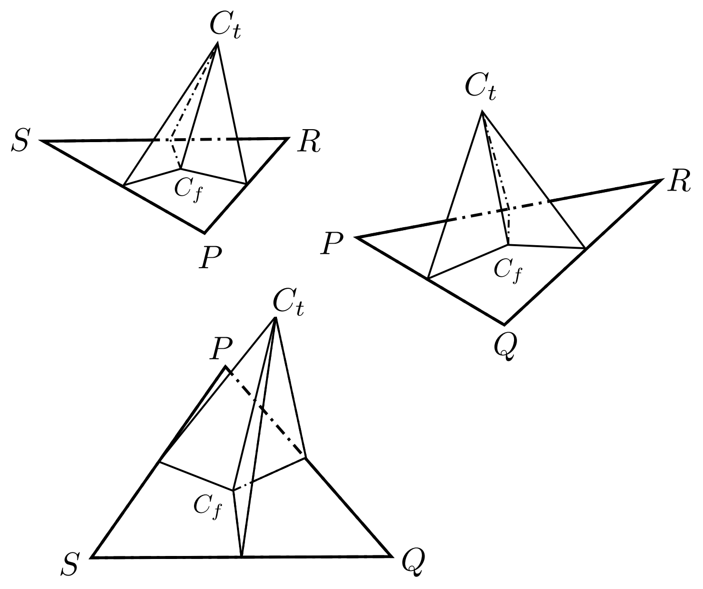


Figure 2.2 Median-dual control volume faces surrounding node 'P' with face centers at C_f

especially in high Reynolds number flows. To handle this, a local coordinate system is created for each tetrahedral element as proposed by Baliga. The local coordinate system is aligned with the local average velocity vector in order improve accuracy.

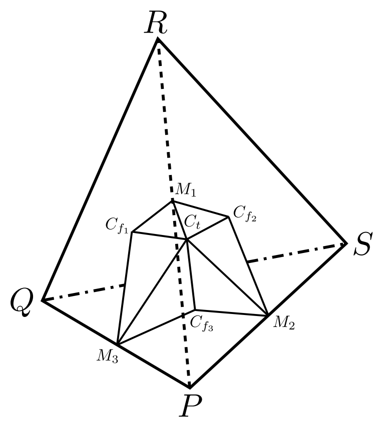


Figure 2.3 Partial median-dual control volume from one tetrahedral element around node 'P', (M_1, M_2, M_3) are edge midpoints

2.3.1 Local Coordinate System

The local coordinate system (X, Y, Z) is transformed so the X direction is aligned with the local average velocity vector $\vec{U}_{avg}$. The Y and Z directions are set so that the local coordinate system is orthogonal. The local coordinate system, illustrated in Fig. 2.4, of a typical tetrahedral control volume with nodes numbered $(1, 2, 3, 4)$. The global coordinate system (x, y, z) is transformed to the local coordinate system (X, Y, Z) using the following steps:

- translate the global origin to the tetrahedral centroid (x_c, y_c, z_c)
- rotate about the y_c axis by θ' to get the temporary axes (x_1, y_1, z_1)
- rotate about the z_1 axis by $(90 - \phi')$ to get the local axes (X, Y, Z)

The transformation angles are given by:

$$\cos(\phi') = \frac{v_{avg}}{U_{avg}} \qquad \cos(\theta') = \frac{u_{avg}}{U_{avg}\sin(\phi')} \tag{2.18}$$

where the average velocities for a tetrahedron with nodes $(1, 2, 3, 4)$ are calculated using:

$$u_{avg} = \frac{(u_1 + u_2 + u_3 + u_4)}{4}; \quad v_{avg} = \frac{(v_1 + v_2 + v_3 + v_4)}{4}; \quad w_{avg} = \frac{(w_1 + w_2 + w_3 + w_4)}{4} \tag{2.19}$$

$$U_{avg} = \sqrt{(u_{avg}{}^2 + v_{avg}{}^2 + w_{avg}{}^2)} \tag{2.20}$$

This results in the following transformation equations:

$$X = \left[(x - x_{ct})\cos(\theta') + (z - z_{ct})\sin(\theta')\right]\sin(\phi') + (y - y_{ct})\cos(\phi') \tag{2.21}$$

$$Y = -\left[(x - x_{ct})\cos(\theta') + (z - z_{ct})\sin(\theta')\right]\cos(\phi') + (y - y_{ct})\sin(\phi') \tag{2.22}$$

$$Z = -(x - x_{ct})\sin(\theta') + (z - z_{ct})\cos(\theta') \tag{2.23}$$

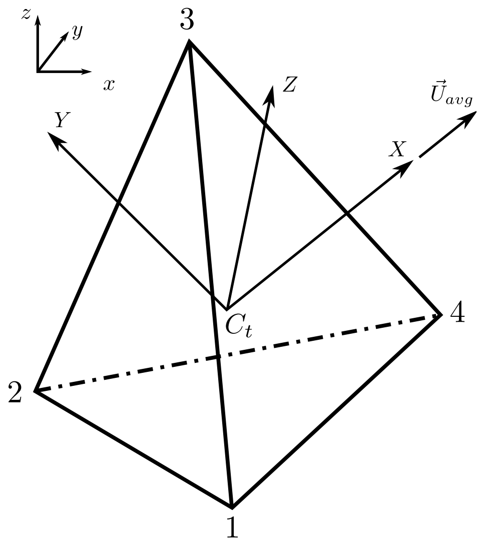


Figure 2.4 Local coordinate system

2.3.2 Interpolation Function for a General Variable ‘Φ’

Baliga [6] formulated an interpolation function for tetrahedral unstructured meshes that uses the exact 1-D, exponential solution while accounting for the three-dimensionality of the flow. Since the exponential function is computationally expensive, the general flow variable Φ is interpolated using the power law scheme in the X direction. The flow variable Φ is assumed to vary linearly in the Y and Z directions. The shape function for a general variable Φ is:

$$\Phi = A\xi + BY + CZ + D \tag{2.24}$$

In the above equation ξ is given by the power law:

$$\xi = \frac{X - X_{max}}{Pe_{\triangle} + [[0, (1 - 0.1|Pe_{\triangle}|^5)]]} \tag{2.25}$$

where the Peclet number $Pe_{\triangle}$ is given by:

$$Pe_{\triangle} = \frac{\rho U_{avg}(X_{max} - Xmin)}{\mu} \tag{2.26}$$

and

$$X_{max} = \max(X_1, X_2, X_3, X_4) \qquad X_{min} = \min(X_1, X_2, X_3, X_4) \tag{2.27}$$

The coefficients A, B, C, D are calculated by satisfying Equation 2.24 at each node point of a tetrahedral element. From Appendix A, the coefficients can be written as:

$$\begin{aligned} A &= L_1W_1 + L_2W_2 + L_3W_3 + L_4W_4 = \sum L_iW_i \\ B &= M_1W_1 + M_2W_2 + M_3W_3 + M_4W_4 = \sum M_iW_i \\ C &= N_1W_1 + N_2W_2 + N_3W_3 + N_4W_4 = \sum N_iW_i \\ D &= O_1W_1 + O_2W_2 + O_3W_3 + O_4W_4 = \sum O_iW_i \end{aligned} \tag{2.28}$$

2.3.3 Flux Computation

Using the interpolation functions and local coordinate system defined above, we can calculate the fluxes J though the control volume faces. The flux equation for a general variable 'Φ' is redefined in the local coordinate system with the velocities u, v and w along the X direction as:

$$J_X^\Phi = \rho u \Phi - \mu \frac{\partial \Phi}{\partial X} \tag{2.29}$$

By substituting the coefficients (Equation 2.28) and the interpolation function (Equation 2.24) into the flux equation, the following equation is obtained:

$$\begin{aligned} J_X^\Phi =& \rho u \left[\left(\sum L_i \Phi_i \right) \xi + \left(\sum M_i \Phi_i \right) Y + \left(\sum N_i \Phi_i \right) Z + \left(\sum O_i \Phi_i \right) \right] \\ & - \mu \left[\left(\sum L_i \Phi_i \right) \left(\frac{\rho U_{avg} \xi}{\mu} + 1 \right) \right] \end{aligned}$$

$$\Rightarrow \boxed{J_X^\Phi = (\rho f_i - \mu L_i) \Phi_i} \tag{2.30}$$

where,

$$f_i = \left[(u - U_{avg})\, L_i \xi + u M_i Y + u N_i Z + u O_i \right]$$

Similarly, the fluxes in the Y and Z directions are:

$$J_Y^\Phi = \rho v \Phi - \mu \frac{\partial \Phi}{\partial Y}$$

$$J_Y^\Phi = \rho v \left[\left(\sum L_i \Phi_i \right) \xi + \left(\sum M_i \Phi_i \right) Y + \left(\sum N_i \Phi_i \right) Z + \left(\sum O_i \Phi_i \right) \right] - \mu \sum M_i \Phi_i$$

$$\Rightarrow \boxed{J_Y^\Phi = (\rho g_i - \mu M_i) \Phi_i} \tag{2.31}$$

where,

$$g_i = v \left[L_i \xi + M_i Y + N_i Z + O_i \right]$$

and

$$J_Z^\Phi = \rho w \Phi - \mu \frac{\partial \Phi}{\partial Z}$$

$$J_Z^\Phi = \rho w \left[\left(\sum L_i \Phi_i \right) \xi + \left(\sum M_i \Phi_i \right) Y + \left(\sum N_i \Phi_i \right) Z + \left(\sum O_i \Phi_i \right) \right] - \mu \sum N_i \Phi_i$$

$$\Rightarrow \boxed{{J_Z}^\Phi = (\rho h_i - \mu N_i) \Phi_i} \tag{2.32}$$

where,

$$h_i = w \left[\xi L_i + M_i Y + N_i Z + O_i \right]$$

A typical median dual face between nodes 1 and 3 is shown in Figure 2.5. The normal direction (n_x, n_y, n_z) for the face is determined by the direction of flow over the tetrahedral face $1 - 2 - 3$. Additionally, the midpoints of the edges of the median dual face are denoted by r, s, t and the area of the face is denoted by A_{rst}. In order to compute the flux through the face, the fluxes $J_X^\Phi, J_Y^\Phi, J_Z^\Phi$ are computed at each midpoint r, s, t. Gauss's quadrature rule is then applied to the midpoint fluxes to compute the flux through the median dual face. The flux of a general variable 'Φ' through the median dual face can be calculated using:

$$\oint\limits_{r-s-t} (\vec{J} \cdot \hat{n}) \mathrm{d}S = \frac{A_{rst}}{3} \left[\left(J_X^r + J_X^s + J_X^t \right) n_x + \left(J_Y^r + J_Y^s + J_Y^t \right) n_y + \left(J_Z^r + J_Z^s + J_Z^t \right) n_z \right] \tag{2.33}$$

The above equation is applied to all the median dual faces that surround the control volume around a node point.

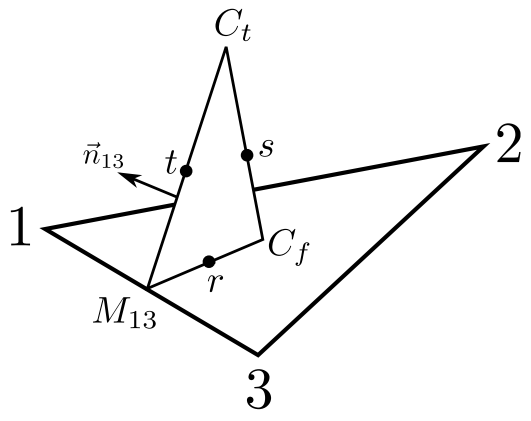


Figure 2.5 Median-dual face between nodes 1 and 3

2.3.4 Interpolation Function for Pressure

As stated earlier, the pressure is assumed to vary linearly within each tetrahedral element. This yields an interpolation function for pressure in global coordinate system of:

$$p = -(\alpha x + \beta y + \gamma z + \eta) \tag{2.34}$$

where the coefficients, α, β, γ and η, are calculated by satisfying the above equation at the node points of the element. The derivation of the coefficients is given in detail in Appendix B. Due to the linear interpolation, the derivatives of the pressure interpolation function are constants. The pressure gradients are simply:

$$-\frac{\partial p}{\partial x} = \alpha \qquad -\frac{\partial p}{\partial y} = \beta \qquad -\frac{\partial p}{\partial z} = \gamma \tag{2.35}$$

Using the derivation in Appendix B, the coefficients and gradients can be written using the pressure at the nodes of the element as:

$$\begin{aligned}
-\frac{\partial p}{\partial x} &= \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \\
-\frac{\partial p}{\partial y} &= \bar{M}_1 p_1 + \bar{M}_2 p_2 + \bar{M}_3 p_3 + \bar{M}_4 p_4 \\
-\frac{\partial p}{\partial z} &= \bar{N}_1 p_1 + \bar{N}_2 p_2 + \bar{N}_3 p_3 + \bar{N}_4 p_4
\end{aligned} \tag{2.36}$$

2.4 Integration and Discretization of the Momentum Equations

Starting with the momentum Equations 2.15-2.17, the equations are integrated using the interpolation functions and fluxes defined in the previous section. The u momentum equation is integrated over a control volume 'd$\forall$' around a representative node 'P'. The v and w momentum equations can be discretized using similar logic. The integral form of the u momentum equation is:

$$\int\limits_{\Delta\forall} \left(\frac{\partial J_x^u}{\partial x} + \frac{\partial J_y^u}{\partial y} + \frac{\partial J_z^u}{\partial z} = S_x' - \frac{\partial p}{\partial x} \right) \mathrm{d}\forall \tag{2.37}$$

where the control volume surrounding node point 'P' has a volume of $\Delta\forall$. For simplification, the above equation can be broken into several parts. The LHS can be written as:

$$\frac{\partial J_x^u}{\partial x} + \frac{\partial J_y^u}{\partial y} + \frac{\partial J_z^u}{\partial z} = \nabla \cdot \vec{J}^u \tag{2.38}$$

The above equation is then rewritten in the local coordinate system as:

$$\nabla \cdot \vec{J}^u = \frac{\partial J_X^u}{\partial X} + \frac{\partial J_Y^u}{\partial Y} + \frac{\partial J_Z^u}{\partial Z} \tag{2.39}$$

The LHS of Equation 2.37 is then replaced with local coordinates in order to reduce false diffusion:

$$\int\limits_{\Delta\forall} \left(\frac{\partial J_X^u}{\partial X} + \frac{\partial J_Y^u}{\partial Y} + \frac{\partial J_Z^u}{\partial Z} \right) \mathrm{d}\forall = \int\limits_{\Delta\forall} \left(S_x' - \frac{\partial p}{\partial x} \right) \mathrm{d}\forall \tag{2.40}$$

Using Gauss's divergence theorem, the volume integral of a general vector $\vec{A}$ can be changed to a surface integral.

$$\int\limits_{\Delta\forall} \left(\nabla \cdot \vec{A} \right) \mathrm{d}\forall = \oint_S \left(\vec{A} \cdot \hat{n} \right) \mathrm{d}S \tag{2.41}$$

where $\hat{n}$ is the normal vector of the control surface. Applying Gauss's theorem to the LHS of Equation 2.40 yields:

$$\int\limits_{\Delta\forall} \left(\frac{\partial J_X^u}{\partial X} + \frac{\partial J_Y^u}{\partial Y} + \frac{\partial J_Z^u}{\partial Z} \right) \mathrm{d}\forall = \int\limits_{\Delta\forall} \left(\nabla \cdot \vec{J}^u \right) \mathrm{d}\forall = \oint \left(\vec{J}^u \cdot \hat{n} \right) \mathrm{d}S \tag{2.42}$$

Since the RHS of Equation 2.40 contains terms that are constant across an element, they can simply be integrated to:

$$\int\limits_{\Delta\forall} \left(S_x' - \frac{\partial p}{\partial x} \right) \mathrm{d}\forall = \left(S_x' - \frac{\partial p}{\partial x} \right) \Delta\forall \tag{2.43}$$

Combining Equations 2.42 and 2.43 yields:

$$\oint \left(\vec{J}^u \cdot \hat{n} \right) \mathrm{d}S = \left(S_x' - \frac{\partial p}{\partial x} \right) \Delta\forall \tag{2.44}$$

2.4.1 Integration of the LHS

Using the Gauss quadrature rule developed in the previous section (Equation 2.33), the LHS of Equation 2.44 can be written as:

$$\oint_{r-s-t} (\vec{J} \cdot \hat{n}) \mathrm{d}S = \frac{A_{rst}}{3} \left[\left(J_X^r + J_X^s + J_X^t \right) n_x + \left(J_Y^r + J_Y^s + J_Y^t \right) n_y + \left(J_Z^r + J_Z^s + J_Z^t \right) n_z \right] \tag{2.45}$$

In the above equation, (n_x, n_y, n_z) are the components of the vector normal to the median-dual surface $(r - s - t)$. For the node 'P', the total flux leaving the control volume is calculated from the contributions of it's neighbors. The fluxes are summed from each of the contributing median-dual faces around node 'P'. Substituting the flux equation and collecting terms for the representative node 'P', the LHS can be simplified to:

$$LHS = a_P u_P - \sum_{i=1}^{N} a_{nb} u_{nb} \tag{2.46}$$

where, 'a_{nb}' represents the terms from neighboring nodes of 'P', the number of neighbor nodes is defined as 'N' and 'a_P' is a collection of terms pertaining to the node 'P':

$$a_P = \sum_{i=1}^{M} a_1^i \tag{2.47}$$

where 'm' is the number of elements that share node 'P'.

2.4.2 Integration of the RHS

As discussed in the previous section, the source terms S'_x and the pressure gradients are assumed to be constant over a tetrahedral element. Since several tetrahedral elements will typically share the node 'P', each will contribute to the source term for the node. The RHS can then be assembled using the contributions of each element that shares node 'P' and the interpolation functions above:

$$RHS = \sum_{i=1}^{m} \left(S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right) \Delta \forall \tag{2.48}$$

where 'm' is the number of tetrahedral elements that share the node 'P'.

2.4.3 Total Discretization Equation

The final u momentum equation about node 'P' can be obtained by combining Equations 2.48 and 2.46:

$$\boxed{a_P u_P = \sum_{i=1}^{N} a_{nb} u_{nb} + \sum_{i=1}^{m} \left(S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right) \Delta\forall} \tag{2.49}$$

where the number of neighbor nodes to point 'P' is defined as 'N' and 'm' is the number of tetrahedral elements that share the point 'P'. Using the same logic, the v and w momentum equations can be developed.

2.4.4 Boundary Condition

To handle prescribed values at boundary faces, the discretized equation is modified. The modifications allow for two types of boundary conditions:

- given Φ (u, v, w)
- given flux F^{Φ}

The modified momentum equation for the boundary is:

$$a_P \Phi_P = \sum_{i}^{N} a_{nb} \Phi_{nb} + \sum_{i=1}^{m} \left(S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right) \Delta\forall - F_P^{\Phi} + \dot{m}_P \Phi_P \tag{2.50}$$

where $\dot{m}_P$ is the total mass flux leaving the domain through the boundary face. If the boundary value of Φ is specified, the boundary flux F_P^{Φ} can be computed using Equation 2.50. Otherwise, if the boundary flux is supplied, then the flow variable Φ can be calculated at the boundary nodes using Equation 2.50.

2.5 Equal Order Velocity-Pressure Interpolation Method

In the previous section, the discretized equations for momentum u, v, w and pressure p were developed. However, for an incompressible flow, no explicit equation for pressure exists since only the pressure gradients appear in the momentum equations. Following the finite volume based SIMPLER approach by Patankar [3], pressure is obtained indirectly through the continuity equation. When the pressure and velocity field are points are collocated at the grid nodes, this can lead to a spurious pressure distribution. One solution used in structured grids is to use a staggered grid formulation in order to eliminate this, however, this approach is not easily implemented in unstructured grids. The unstructured formulation for pressure use the velocity-pressure method developed by Prakash and Patankar [7]. The general idea behind the method is that the velocity field used to solve the continuity equation should be dependent on the pressure differences between adjacent nodes. This new velocity, known as the "artificial velocity", no longer allows spurious checkerboard pressure fields.

2.5.1 Definition of Pseudo Velocities $(\hat{u}, \hat{v}, \hat{w})$ and Source Term Coefficients (d^u, d^v, d^w)

Before developing the artificial velocity, we must first introduce the pseudo velocity and source term coefficients. As in previous sections, the derivation will be shown in the x direction and the y and z directions can be derived using the same logic. By rearranging the momentum Equation 2.49, the u velocity can be written as:

$$u_P = \frac{\sum_{i=1}^{N}(a_{nb}u_{nb})}{a_P^u} + \frac{\Delta\forall(S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4)}{a_P^u} \tag{2.51}$$

The pseudo velocity in the u direction at node 'P' is then defined as:

$$\hat{u}_P = \frac{\sum_{i=1}^{N}(a_{nb}u_{nb})}{a_P^u} \tag{2.52}$$

where 'N' stands for the number of neighboring nodes around node 'P'. Additionally, the source term coefficient can be defined as:

$$d_P^u = \frac{\Delta\forall}{a_P^u} \tag{2.53}$$

In the above equation, $\Delta\forall$ is the control volume surrounding node 'P'. The pseudo velocities $(\hat{u}, \hat{v}, \hat{w})$ and the source term coefficients (d^u, d^v, d^w) are assumed to vary linearly within a tetrahedral element.

2.5.2 Definition of the Artificial Velocity Field

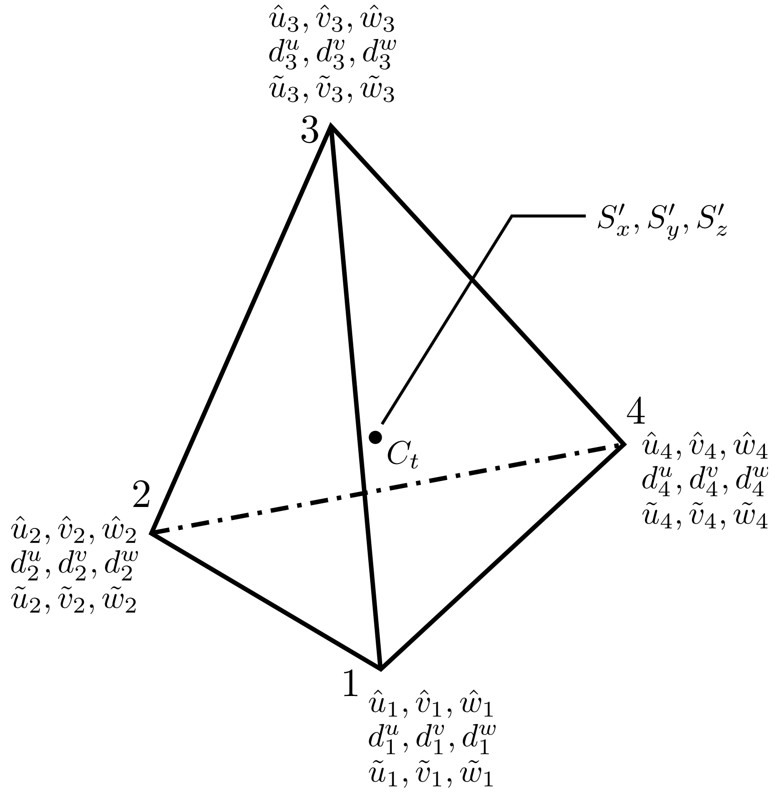

$\hat{u}_3, \hat{v}_3, \hat{w}_3$
d_3^u, d_3^v, d_3^w
$\tilde{u}_3, \tilde{v}_3, \tilde{w}_3$
3
S_x', S_y', S_z'
4
$\hat{u}_4, \hat{v}_4, \hat{w}_4$
d_4^u, d_4^v, d_4^w
$\tilde{u}_4, \tilde{v}_4, \tilde{w}_4$
C_t
2
$\hat{u}_2, \hat{v}_2, \hat{w}_2$
d_2^u, d_2^v, d_2^w
$\tilde{u}_2, \tilde{v}_2, \tilde{w}_2$
1
$\hat{u}_1, \hat{v}_1, \hat{w}_1$
d_1^u, d_1^v, d_1^w
$\tilde{u}_1, \tilde{v}_1, \tilde{w}_1$

Figure 2.6 Interpolation of artificial velocity in a tetrahedral element

For the equal-order method, the artificial velocity ($\vec{\tilde{U}}$) is defined for a tetrahedral element using the following equation:

$$\vec{\tilde{U}} = \tilde{u}\hat{i} + \tilde{v}\hat{j} + \tilde{w}\hat{k} \tag{2.54}$$

where the artificial velocity components are:

$$\begin{aligned}
\tilde{u} =& \hat{u} + d^u \left(S_x' - \frac{\partial p}{\partial x} \right) \\
\tilde{v} =& \hat{v} + d^v \left(S_y' - \frac{\partial p}{\partial y} \right) \\
\tilde{w} =& \hat{w} + d^w \left(S_z' - \frac{\partial p}{\partial z} \right)
\end{aligned} \tag{2.55}$$

Combining the pseudo velocity and source term coefficient definitions with the artificial velocity components for a node 'P' yields:

$$\tilde{u}_P = \hat{u}_P + d_P^u \sum_{i=1}^{m} (S'_x + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4) \tag{2.56}$$

$$\tilde{v}_P = \hat{v}_P + d_P^v \sum_{i=1}^{m} (S'_y + \bar{M}_1 p_1 + \bar{M}_2 p_2 + \bar{M}_3 p_3 + \bar{M}_4 p_4) \tag{2.57}$$

$$\tilde{w}_P = \hat{w}_P + d_P^w \sum_{i=1}^{m} (S'_z + \bar{N}_1 p_1 + \bar{N}_2 p_2 + \bar{N}_3 p_3 + \bar{N}_4 p_4) \tag{2.58}$$

In the above equation, it can be seen clearly that the artificial velocity is dependent on the surrounding pressure values. This artificial velocity is used to satisfy the continuity criteria instead of the nodal velocity and the resulting equation is used to solve for the pressure. This pressure driven velocity ensures that spurious pressure fields such as checkerboard patterns are not permissible solutions. Since the mass conservation equation is now being solved with the artificial velocity, the coefficient for the momentum equations must also be found using the artificial velocity in order to preserve overall conservation. Since the derivation of the momentum coefficients was already shown in a previous section, the next section will show how these can be recast to use the artificial velocity.

2.5.3 Interpolation of $\vec{\tilde{U}}$ at the Control Volume Faces

In order to recast the coefficients of the momentum equations using the artificial velocities, the same steps are taken except the nodal velocities are replaced with the artificial velocities. Also the local coordinate system (Equations 2.19 - 2.21) are recast using the artificial velocities $(\hat{u}, \hat{v}.\hat{w})$ instead of the nodal velocities (u, v, w). In order to calculate the fluxes, the artificial velocities need to be found at the edge midpoints (r, s, t) (Figure 2.5). Since the artificial velocities and source term coefficients are assumed to vary linearly across each tetrahedral element and the pressure derivatives and source terms are assumed to be constant across each element, the artificial velocity will vary linearly across an element. Since we can use a linear interpolation, we first need the artificial velocities at each of the node points that make up the face $(1 - 2 - 3)$.

$$\begin{aligned}
\tilde{u}_1 =& \hat{u}_1 + d_1^u \sum_{i=1}^{m} \left(S'_x + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right) \\
\tilde{u}_2 =& \hat{u}_2 + d_2^u \sum_{i=1}^{m} \left(S'_x + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right) \\
\tilde{u}_3 =& \hat{u}_3 + d_3^u \sum_{i=1}^{m} \left(S'_x + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right) \qquad (2.59)
\end{aligned}$$

Using a simple linear interpolation, the artificial velocity at the center of the tetrahedron '$\tilde{u}_c$' and at the center of the face '$\tilde{u}_{cf}$' can be calculated using:

$$\tilde{u}_{cf} = \frac{\tilde{u}_1 + \tilde{u}_2 + \tilde{u}_3}{3} \qquad (2.60)$$

$$\tilde{u}_{ct} = \frac{\tilde{u}_1 + \tilde{u}_2 + \tilde{u}_3 + \tilde{u}_4}{4} \qquad (2.61)$$

Using the artificial velocities at the face vertices, face center and the center of the tetrahedron, the artificial velocities can be calculated at the center of the edges ($\vec{\tilde{U}}_r, \vec{\tilde{U}}_s, \vec{\tilde{U}}_t$):

$$\vec{\tilde{U}}_r = \frac{5\left(\vec{\tilde{U}}_1 + \vec{\tilde{U}}_3\right) + 2\vec{\tilde{U}}_2}{12} \qquad (2.62)$$

$$\vec{\tilde{U}}_s = \frac{7\left(\vec{\tilde{U}}_1 + \vec{\tilde{U}}_2 + \vec{\tilde{U}}_3\right) + 3\vec{\tilde{U}}_4}{24} \qquad (2.63)$$

$$\vec{\tilde{U}}_t = \frac{3\left(\vec{\tilde{U}}_1 + \vec{\tilde{U}}_3\right) + \left(\vec{\tilde{U}}_2 + \vec{\tilde{U}}_4\right)}{8} \qquad (2.64)$$

The fluxes are summed from each of the contributing median-dual faces around node 'P' and the artificial velocities are then used to compute the momentum coefficients using Equations 2.45 - 2.49.

2.6 Pressure Equation

The pressure equation can now be derived using the artificial velocity field $\vec{\tilde{U}}$ form the continuity equation. The continuity equation written for the a median-dual face in terms of the artificial velocity can be written as:

$$\oint_{r-s-t} \rho \vec{\tilde{U}}_{cf} \cdot \hat{n}\, \mathrm{d}S = 0 \tag{2.65}$$

Expanding the equation in terms of the velocities at the midpoint of the face edges $(r-s-t)$, the equation becomes:

$$\rho \left[\left(\frac{\tilde{u}_r + \tilde{u}_s + \tilde{u}_t}{3} \right) n_x + \left(\frac{\tilde{v}_r + \tilde{v}_s + \tilde{v}_t}{3} \right) n_y + \left(\frac{\tilde{w}_r + \tilde{w}_s + \tilde{w}_t}{3} \right) n_z \right] = 0 \tag{2.66}$$

Substituting the artificial velocities and expanding the above equation, the pressure equation can be derived. Applying the equation to a general node 'Q' and collecting terms, the pressure equation can be written as:

$$\boxed{a_Q^p p_Q = \sum_{i=1}^{N} \left(a_{nb}^p p_{nb} \right) + b_p} \tag{2.67}$$

In the above equation a_{nb}^p represents the coefficients for each of the neighboring nodes and a_Q^p is the collection of terms representing node 'Q'. Any remaining terms make up the source term for the pressure equation b_p:

$$a_i^p = \rho \left[\left(d_i^u \bar{L}_i \right) n_x + \left(d_i^v \bar{M}_i \right) n_y + \left(d_i^w \bar{N}_i \right) n_z \right] \tag{2.68}$$

$$b_p = \sum_{i=1}^{m} \rho \left[\left(\hat{u}_i + d_i^u S^u \right) n_x + \left(\hat{v}_i + d_i^v S^y \right) n_y + \left(\hat{w}_i + d_i^w S^w \right) n_z \right] \tag{2.69}$$

A detailed derivation is shown in Appendix C.

2.6.1 Pressure Equation for the Boundaries

In order to account for the mass flow leaving the boundary face, the discretized pressure equation needs to be modified. This is accomplished by modifying Equation 2.67 into the following equation:

$$a_Q^p p_Q = \sum_{i=1}^{N} \left(a_{nb}^p\, p_{nb} \right) + b_p - \dot{m}_Q \tag{2.70}$$

where $\dot{m}_Q$ is the mass flux leaving the boundary face and point 'Q' is a boundary node.

2.7 Pressure Correction

In general, the velocities obtained by solving the momentum equations will not satisfy the continuity equation and is thus, not converged. The convergence can be accelerated through the use of a pressure correction equation that will be used to correct the velocities in order to satisfy the conservation equation at each iteration. The velocity correction is especially useful when using the equal order method, since the conservation equations are solved using the artificial velocities and do not directly affect the nodal velocities. This technique was popularized in the SIMPLE and SIMPLER algorithms and a similar approach is adopted here.

If we denote the pressure field and velocities solve for using the previous equations with a star (p^*, u^*, v^*, w^*) we can rewrite the artificial velocities as:

$$\begin{aligned}
\tilde{u}^* &= \hat{u}^* + d^u\left(S'_x - \frac{\partial p^*}{\partial x}\right)\\
\tilde{v}^* &= \hat{v}^* + d^v\left(S'_y - \frac{\partial p^*}{\partial y}\right)\\
\tilde{w}^* &= \hat{w}^* + d^w\left(S'_z - \frac{\partial p^*}{\partial z}\right)
\end{aligned} \tag{2.71}$$

If the pressure correction field p' is added to the original pressure field p^* then the corrected pressure will be given by:

$$p = p^* + p' \tag{2.72}$$

Using the corrected pressure, the artificial velocities can be redefined so that they satisfy the continuity equation:

$$\begin{aligned}
\tilde{u} =& \hat{u}^* + d^u \left(S'_x - \frac{\partial \left(p^* + p' \right)}{\partial x} \right) \\
\tilde{v} =& \hat{v}^* + d^v \left(S'_y - \frac{\partial \left(p^* + p' \right)}{\partial y} \right) \\
\tilde{w} =& \hat{w}^* + d^w \left(S'_z - \frac{\partial \left(p^* + p' \right)}{\partial z} \right)
\end{aligned} \tag{2.73}$$

Defining the velocities in a similar form as the corrected pressure Equation 2.72 we get:

$$\begin{aligned}
\tilde{u} =& \tilde{u}^* + \tilde{u}' \\
\tilde{v} =& \tilde{v}^* + \tilde{v}' \\
\tilde{w} =& \tilde{w}^* + \tilde{w}'
\end{aligned} \tag{2.74}$$

Using Equations 2.73 and the velocity correction terms are given by:

$$\begin{aligned}
\tilde{u}' =& - d^u \left(\frac{\partial p'}{\partial x} \right) \\
\tilde{v}' =& - d^v \left(\frac{\partial p'}{\partial y} \right) \\
\tilde{w}' =& - d^w \left(\frac{\partial p'}{\partial z} \right)
\end{aligned} \tag{2.75}$$

By comparing the pressure equation with the correction equations above, the only difference is the source term. The source terms for the pressure correction equation are:

$$\begin{aligned}
S_p'^u =& S'_x - \frac{\partial p^*}{\partial x} \\
S_p'^v =& S'_y - \frac{\partial p^*}{\partial y} \\
S_p'^w =& S'_z - \frac{\partial p^*}{\partial z}
\end{aligned} \tag{2.76}$$

Applying the above equations, the pressure correction equation around node 'Q' can be reduced to a similar form as the pressure equation:

$$a'_Q p'_Q = \sum_{i=1}^{N} (a'_{nb} \, p'_{nb}) + b'_p \tag{2.77}$$

Similar to the pressure equation, the pressure correction equation is modified for a boundary node to account for the mass flux leaving the boundary face:

$$a'_Q p'_Q = \sum_{i=1}^{N} (a'_{nb} \, p'_{nb}) + b'_p - \dot{m}_Q \tag{2.78}$$

In the above equations, a'_Q and a'_{nb} are the same coefficients used in the pressure equation. Once the equations are solved for the pressure correction field p', the corrected artificial velocity $\vec{\tilde{U}}'$ can be calculated. The velocity correction formula developed by Prakash [5] is then used to get the nodal velocity correction terms:

$$\begin{aligned}
u'_Q &= \frac{-d^u_Q}{\Delta\forall_Q} \sum_{i=1}^{m} \Delta\forall_i \left(\frac{\partial p'}{\partial x}\right)_i \\
v'_Q &= \frac{-d^v_Q}{\Delta\forall_Q} \sum_{i=1}^{m} \Delta\forall_i \left(\frac{\partial p'}{\partial y}\right)_i \\
w'_Q &= \frac{-d^w_Q}{\Delta\forall_Q} \sum_{i=1}^{m} \Delta\forall_i \left(\frac{\partial p'}{\partial z}\right)_i
\end{aligned} \tag{2.79}$$

The above equation is formulated for a general point 'Q' which is part of the m^{th} tetrahedral element and has a surrounding control volume of $\Delta\forall$. Finally, the nodal velocities are corrected using the above equations and the nodal velocities calculated using the momentum equations:

$$\begin{aligned}
u_Q &= u^*_Q + u'_Q \\
v_Q &= v^*_Q + v'_Q \\
w_Q &= w^*_Q + w'_Q
\end{aligned} \tag{2.80}$$

The velocities are not corrected on the boundaries since this would change the prescribed boundary condition.

2.8 Relaxation

The discretized equations for the momentum equations, pressure equation and pressure correction equation are solved using Gauss-Seidel iterations with alternating direction sweeps.

The convergence can be accelerated by using relaxation on the discretized equations however, relaxation should not be applied to the pressure correction equation. Applying a relaxation factor β' to the equation for a general variable 'Φ' yields:

$$a_P \Phi_P = \beta' \left[\sum_{i=1}^{N} a_{nb}\, \Phi_{nb} + \sum_{i=1}^{m} \left(S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right) \right] + \left(1 - \beta'\right) a_P \Phi_P^o \quad (2.81)$$

From previous research [3], it was discovered that for SIMPLER based algorithms, under-relaxation is required to accelerate the convergence. In order to be under-relaxed, the relaxation factor β' needs to be less than one and is preferably a small value. For this research, a relaxation factor of $\beta' = 0.05$ is used.

CHAPTER 3. TIME INTEGRATION SCHEMES

In this research three different time integration schemes are used, Fully Implicit, Crank-Nicolson and Runge-Kutta. The first two schemes follow the traditional SIMPLER algorithm outlined by Patanker [3] for the pressure-velocity coupling along with a pressure correction equation to conserve mass at each iteration. In the Runge-Kutta scheme the discretized equation is obtained as discussed in the previous chapter as is the case for Fully Implicit and Crank-Nicholson schemes. However, a four-stage Runge-Kutta algorithm is used to update the velocity components without the use of the pressure correction equation. While the Fully-Implicit and Crank-Nicolson schemes require an iterative process during which the velocities are updated at every sub iteration; the Runge-Kutta scheme only updates the velocity once before advancing to the next time step. The Fully Implicit, Crank-Nicolson and Runge-Kutta algorithms are described in this section.

3.1 Time Integration Method

3.1.1 Fully Implicit and Crank-Nicolson Method

If the general variable Φ is to be integrated, it is necessary to make an assumption as to how it varies with time. Although there are many options, one possibility is to use the following stencil:

$$\int_t^{t+\Delta t} \Phi \, \mathrm{d}t = [\alpha_t \Phi + (1-\alpha_t)\Phi^0]\Delta t \tag{3.1}$$

Where α_t is the weighting factor, Φ is the value at time $t + \Delta t$ and Φ^0 is the value at t. By setting the weighting factor $\alpha = 0.5$, the integration yields the Crank-Nicolson scheme and by using $\alpha_t = 1$ yields the Fully Implicit scheme.

3.1.2 General Convection-Diffusion Equation

The three-dimensional General Convection-Diffusion equation can be written as:

$$\frac{\partial(\rho\Phi)}{\partial t} + \frac{\partial J_x}{\partial x} + \frac{\partial J_y}{\partial y} + \frac{\partial J_z}{\partial z} = S_\Phi \tag{3.2}$$

Where S_Φ, J_x, J_y and J_z are the source term and total (convection plus diffusion) fluxes respectively. The total fluxes can be defined by:

$$J_x \equiv \rho u\Phi - \mu\frac{\partial\Phi}{\partial x} \qquad J_y \equiv \rho v\Phi - \mu\frac{\partial\Phi}{\partial y} \qquad J_z \equiv \rho w\Phi - \mu\frac{\partial\Phi}{\partial z} \tag{3.3}$$

Integrating Equation 3.2 in the x direction over the control volume and with respect to time can be written as:

$$\iint\limits_{\Delta\forall} \frac{\partial\rho u}{\partial t}\mathrm{d}t\mathrm{d}\forall + \iint\limits_{\Delta\forall} \left(\frac{\partial J_x^u}{\partial x} + \frac{\partial J_y^u}{\partial y} + \frac{\partial J_z^u}{\partial z}\right)\mathrm{d}\forall\mathrm{d}t = \iint\limits_{\Delta\forall}\left(S_x - \frac{\partial p}{\partial x}\right)\mathrm{d}\forall\mathrm{d}t \tag{3.4}$$

The first term on the LHS can be expanded to:

$$\iint\limits_{\Delta\forall} \frac{\partial\rho u}{\partial t}\mathrm{d}t\mathrm{d}\forall = \left[(\rho u) - (\rho u)^o\right]\Delta\forall \tag{3.5}$$

Applying Gauss-Divergence theorem to the second term in the LHS yields:

$$\iint\limits_{\Delta\forall} \left(\frac{\partial J_x^u}{\partial x} + \frac{\partial J_y^u}{\partial y} + \frac{\partial J_z^u}{\partial z}\right)\mathrm{d}\forall\mathrm{d}t = \int\oint\left(\vec{J}^u\cdot\hat{n}\right)\mathrm{d}S\mathrm{d}t =$$

$$\Delta t\left[f\left(a_P u_P - \sum_{i=1}^{N} a_{nb}u_{nb}\right) + (1-f)\left(a_P u_P - \sum_{i=1}^{N} a_{nb}u_{nb}\right)^o\right] \tag{3.6}$$

Integration of the RHS results in:

$$\iint\limits_{\Delta\forall}\left(S_x - \frac{\partial p}{\partial x}\right)\mathrm{d}\forall\mathrm{d}t = \Delta t\Delta\forall\sum_{i=1}^{m}[f\left(S'_X + \bar{L}_1p_1 + \bar{L}_2p_2 + \bar{L}_3p_3 + \bar{L}_4p_4\right)$$

$$+(1-f)\left(S'_X + \bar{L}_1p_1 + \bar{L}_2p_2 + \bar{L}_3p_3 + \bar{L}_4p_4\right)^o] \tag{3.7}$$

The total discretized equation can be simplified as:

$$a_P u_P = f[\sum_{i=1}^{N} a_{nb}u_{nb}] + b_u + b_u^o \tag{3.8}$$

where;

$$a_P = \frac{\rho \Delta \forall}{\Delta t} + f a_P \tag{3.9}$$

$$b_u = f \sum_{i=1}^{m} \left[S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right] \Delta \forall \tag{3.10}$$

$$b_u^o = \left[\frac{\rho \Delta \forall}{\Delta t} - (1-f) a_P \right] u_P^o + (1-f) [\sum_{i=1}^{N} a_{nb} u_{nb}]^o \tag{3.11}$$

$$+ (1-f) \Delta \forall \sum_{i=1}^{m} \left[S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right]^o \tag{3.12}$$

In Equations 3.8 through 3.12, 'N' stands for the number of neighboring nodes around grid point 'P' and 'm' is the number of tetrahedral cells of which node 'P' is a part of. A similar process can be used to derive the equations for u and v.

3.1.3 Explicit Runge-Kutta Method

The spatial discretization of the Navier-Stokes equations change the system of partial differential equations into a coupled set of ordinary differential equations. To integrate the general variable Φ, the following definition can be used:

$$\frac{\mathrm{d}\Phi}{\mathrm{d}t} = \frac{R(\Phi)}{\rho \Delta \forall} \tag{3.13}$$

A four-stage Runge-Kutta scheme is then applied using the equation above:

$$\Phi^{(0)} = \Phi^n \tag{3.14}$$

$$\Phi^{(1)} = \Phi^{(0)} + \frac{1}{4} \frac{\Delta t}{\rho \Delta \forall} R(\Phi^{(0)}) \tag{3.15}$$

$$\Phi^{(2)} = \Phi^{(0)} + \frac{1}{3} \frac{\Delta t}{\rho \Delta \forall} R(\Phi^{(1)}) \tag{3.16}$$

$$\Phi^{(3)} = \Phi^{(0)} + \frac{1}{2} \frac{\Delta t}{\rho \Delta \forall} R(\Phi^{(2)}) \tag{3.17}$$

$$\Phi^{(4)} = \Phi^{(0)} + \frac{\Delta t}{\rho \Delta \forall} R(\Phi^{(3)}) \tag{3.18}$$

$$\Phi^{n+1} = \Phi^{(4)} \tag{3.19}$$

Equation 3.13 written for the different directions yields:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \frac{R_u}{\rho \Delta \forall} \tag{3.20}$$

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{R_v}{\rho \Delta \forall} \tag{3.21}$$

$$\frac{\mathrm{d}w}{\mathrm{d}t} = \frac{R_w}{\rho \Delta \forall} \tag{3.22}$$

The conservation form of the momentum equation for the u velocity is:

$$\frac{\mathrm{d}(\rho u)}{\mathrm{d}t} \Delta \forall + \oint \left(\vec{J}^u \cdot \hat{n} \right) \mathrm{d}S = \iiint \left[S'_x - \frac{\partial p}{\partial x} \right] \mathrm{d}\forall \tag{3.23}$$

By following the spatial discretization in Chapter 2, the equation can be rearranged and expressed as follows:

$$\frac{\mathrm{d}(\rho u)}{\mathrm{d}t} \Delta \forall = - \sum a_i u_i + \left[S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right] \Delta \forall \tag{3.24}$$

Collecting terms and using a general point P and it's neighbors, the equation can be arranged as:

$$\frac{\mathrm{d}(\rho u)}{\mathrm{d}t} \Delta \forall = \sum_{(i=1)}^{N} a_{nb} u_{nb} - a_P u_P + \left[S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right] \Delta \forall \tag{3.25}$$

Letting,

$$b_u = \left[S'_X + \bar{L}_1 p_1 + \bar{L}_2 p_2 + \bar{L}_3 p_3 + \bar{L}_4 p_4 \right] \Delta \forall \tag{3.26}$$

and assuming incompressible flow, Equation 3.25 can be rewritten as:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \frac{\left(\sum_{(i=1)}^{N} a_{nb} u_{nb} - a_P u_P + b_u \right)}{\rho \Delta \forall} \tag{3.27}$$

Using Equation 3.13, the residual function for the u velocity can be obtained:

$$R_u = \sum_{(i=1)}^{N} a_{nb} u_{nb} - a_P u_P + b_u \tag{3.28}$$

Similarly, the v and w momentum equations can be obtained as shown below:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{\left(\sum_{(i=1)}^{N} a_{nb}v_{nb} - a_P v_P + b_v\right)}{\rho\Delta\forall} \tag{3.29}$$

$$b_v = \left[S'_Y + \bar{M}_1 p_1 + \bar{M}_2 p_2 + \bar{M}_3 p_3 + \bar{M}_4 p_4\right]\Delta\forall \tag{3.30}$$

$$R_v = \sum_{(i=1)}^{N} a_{nb}v_{nb} - a_P v_P + b_v \tag{3.31}$$

$$\frac{\mathrm{d}w}{\mathrm{d}t} = \frac{\left(\sum_{(i=1)}^{N} a_{nb}w_{nb} - a_P w_P + b_w\right)}{\rho\Delta\forall} \tag{3.32}$$

$$b_w = \left[S'_Z + \bar{N}_1 p_1 + \bar{N}_2 p_2 + \bar{N}_3 p_3 + \bar{N}_4 p_4\right]\Delta\forall \tag{3.33}$$

$$R_w = \sum_{(i=1)}^{N} a_{nb}w_{nb} - a_P w_P + b_w \tag{3.34}$$

3.2 Solution Procedure

3.2.1 SIMPLER Algorithm

In general, SIMPLER is typically used with a Fully-Implicit or Crank-Nicolson time integration scheme as described in Section 3.1. The solution procedure for the discretized equations is as follows:

1. Initialize the values of u, v, w and p at all grid points or use the values from the previous time step.

2. Calculate the artificial velocity $(\tilde{u}, \tilde{v}, \tilde{w})$ using Equations 2.56 through 2.58

3. Calculate and assemble the coefficients for the momentum equations.

4. Calculate the unsteady terms and modify the momentum coefficients from Step 3 using Equations 3.9 through 3.12.

5. Using the nodal velocities and momentum coefficients, calculate the pseudo velocities $(\hat{u}, \hat{v}, \hat{w})$ and source term coefficients (d^u, d^v, d^w) using Equations 2.52 and 2.53.

6. Calculate the coefficients and source terms for the pressure equation.

7. Using the pressure coefficients and source terms from Step 6, solve for the pressure field p^* using Equation 2.67.

8. With the pressure field p^* and the momentum coefficients from Step 3, solve the momentum equations for the nodal velocities (u^*, v^*, w^*) and update the fluxes at the boundary points (F^u, F^v, F^w).

9. Recalculate the pseudo velocities $(\hat{u}^*, \hat{v}^*, \hat{w}^*)$ using the new nodal velocities (u^*, v^*, w^*).

10. Calculate the source term for the pressure correction equation.

11. Solve the pressure correction equation to obtain p'.

12. Correct the velocity components (u, v, w) using the pressure correction field p'.

13. Return to Step 3 and repeat until convergence.

14. Proceed to the next time step using $(u, v, w$ and $p)$ as the initial guess in Step 1.

Since the equations are non-linear, an iterative process needs to be used. In order to acceletrate convergene, the pressure and momentum equations may need to be under-relaxed. The pressure correction equation should not be relaxed since, it is used to correct the velocity field to conserve mass. In order to satisfy continuity during each iteration, the solution to the correction equation needs to be fairly well converged.

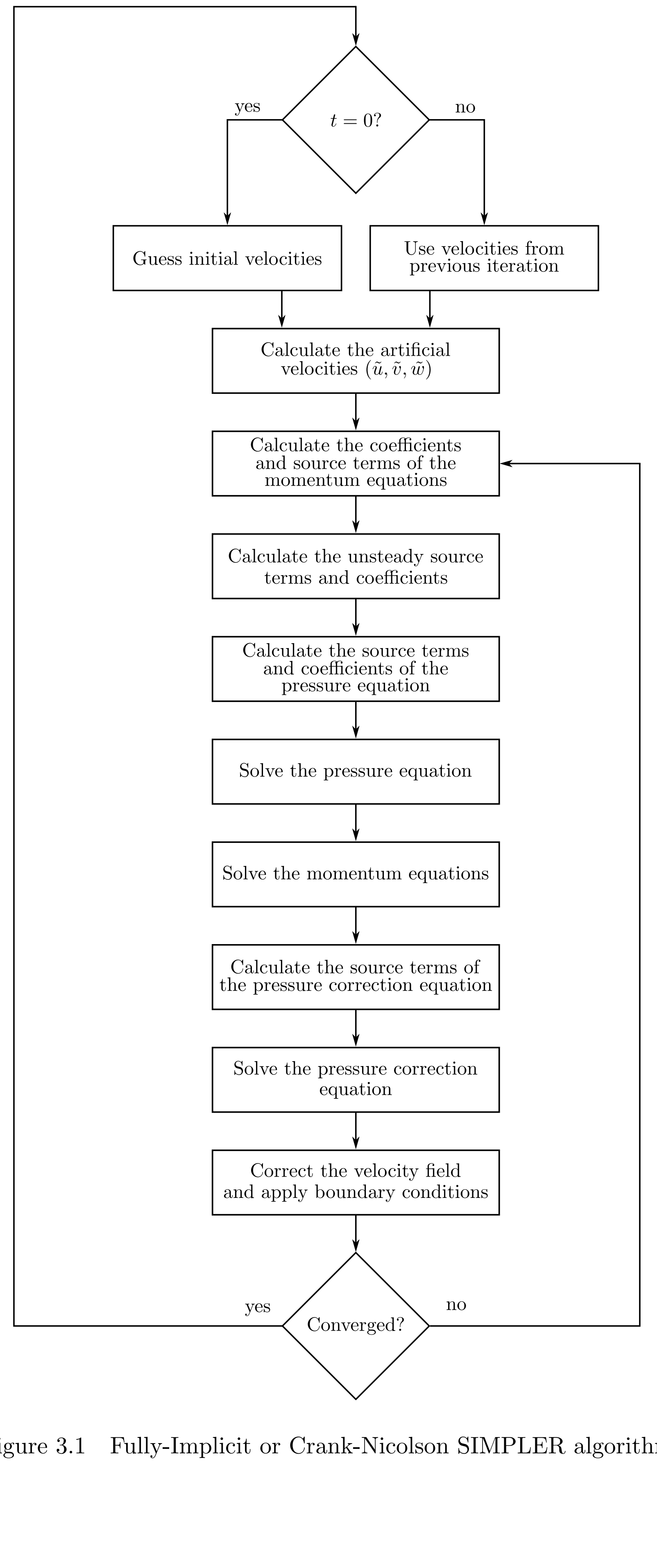

$t = 0?$
yes
no
Guess initial velocities
Use velocities from previous iteration
Calculate the artificial velocities $(\tilde{u}, \tilde{v}, \tilde{w})$
Calculate the coefficients and source terms of the momentum equations
Calculate the unsteady source terms and coefficients
Calculate the source terms and coefficients of the pressure equation
Solve the pressure equation
Solve the momentum equations
Calculate the source terms of the pressure correction equation
Solve the pressure correction equation
Correct the velocity field and apply boundary conditions
yes
Converged?
no

Figure 3.1 Fully-Implicit or Crank-Nicolson SIMPLER algorithm.

3.2.2 Runge-Kutta Algorithm for Low Speed Flows

The Runge-Kutta SIMPLER algorithm for low speed flows follows:

1. Initialize the values of u, v, w and p at all grid points or use the values from the previous time step.

2. Calculate the artificial velocity $(\tilde{u}, \tilde{v}, \tilde{w})$ using Equations 2.56 through 2.58

3. Calculate and assemble the coefficients for the momentum equations.

4. Calculate the unsteady terms and modify the momentum coefficients from Step 3 using Equations 3.9 through 3.12.

5. Using the nodal velocities and momentum coefficients, calculate the pseudo velocities $(\hat{u}, \hat{v}, \hat{w})$ and source term coefficients (d^u, d^v, d^w) using Equations 2.52 and 2.53.

6. Calculate the coefficients and source terms for the pressure equation.

7. Using the pressure coefficients and source terms from Step 6, solve for the pressure field p^* using Equation 2.67.

8. With the pressure field p^* as the source, use the four-stage Runge-Kutta algorithm to update the nodal velocities (u, v, w) (Equations 3.14 - 3.19). The residuals are calculated using Equations 3.28, 3.31 and 3.34 with the coefficients and source terms held constant without the unsteady terms during this process.

9. Proceed to the next time step using $(u, v, w$ and $p)$ as the initial guess in Step 1.

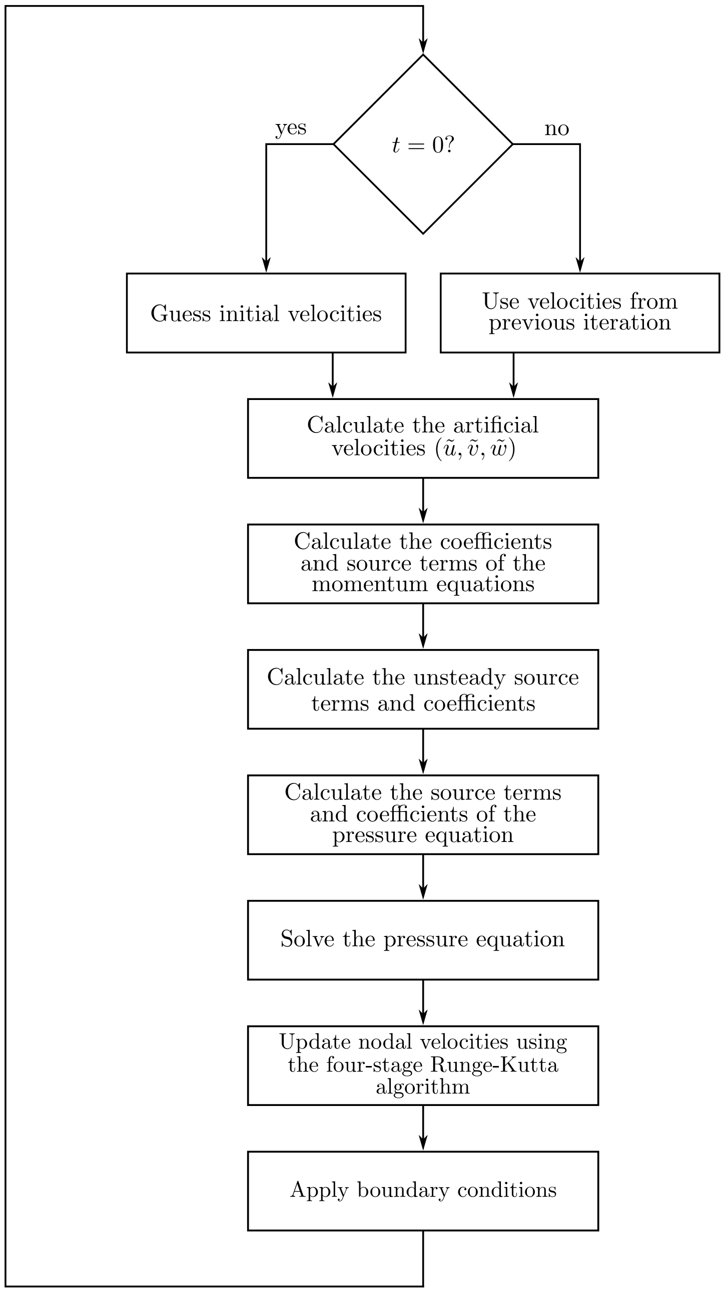

$t = 0$?
yes
no
Guess initial velocities
Use velocities from previous iteration
Calculate the artificial velocities $(\tilde{u}, \tilde{v}, \tilde{w})$
Calculate the coefficients and source terms of the momentum equations
Calculate the unsteady source terms and coefficients
Calculate the source terms and coefficients of the pressure equation
Solve the pressure equation
Update nodal velocities using the four-stage Runge-Kutta algorithm
Apply boundary conditions

Figure 3.2 Four-stage Runge-Kutta SIMPLER algorithm.

CHAPTER 4. GPU IMPLEMENTATION OF CFD

The current trend in graphics card performance can be directly attributed to the demand for increasingly realistic visual effects in video games. This market demand has created an intense rivalry between NVIDIA and ATI (the two primary GPU manufacturers). By design, graphics processing is a highly parallel, data intensive process. The high demand for GPU with more raw parallel computation power and the rivalry between companies has spurred a rapid increase in GPU power that has far outpaced the advances in the CPU market (Figure 4.1). Early attempts at harnessing this power for computational purposes involved, mapping equations to graphics functions in order to trick the GPU. This proved the concept of GPGPU computing but was very time consuming to develop programs. In the early 2000s, this method was replaced by Stanford's *BrookGPU* which allowed programmers to utilize traditional C like programming to utilize the graphics card processor without extensive graphics-specific knowledge. With GPGPU computing becoming more popular, several companies have released languages specifically for use with graphics cards. These include NVIDIA's *CUDA*, ATI's *Stream*, Microsoft's *DirectCompute* and the *OpenCL* framework. It is the author's opinion that NVIDIA's graphics hardware and CUDA software provide the highest performance and flexibility at the time of writing, and is the focus of this research.

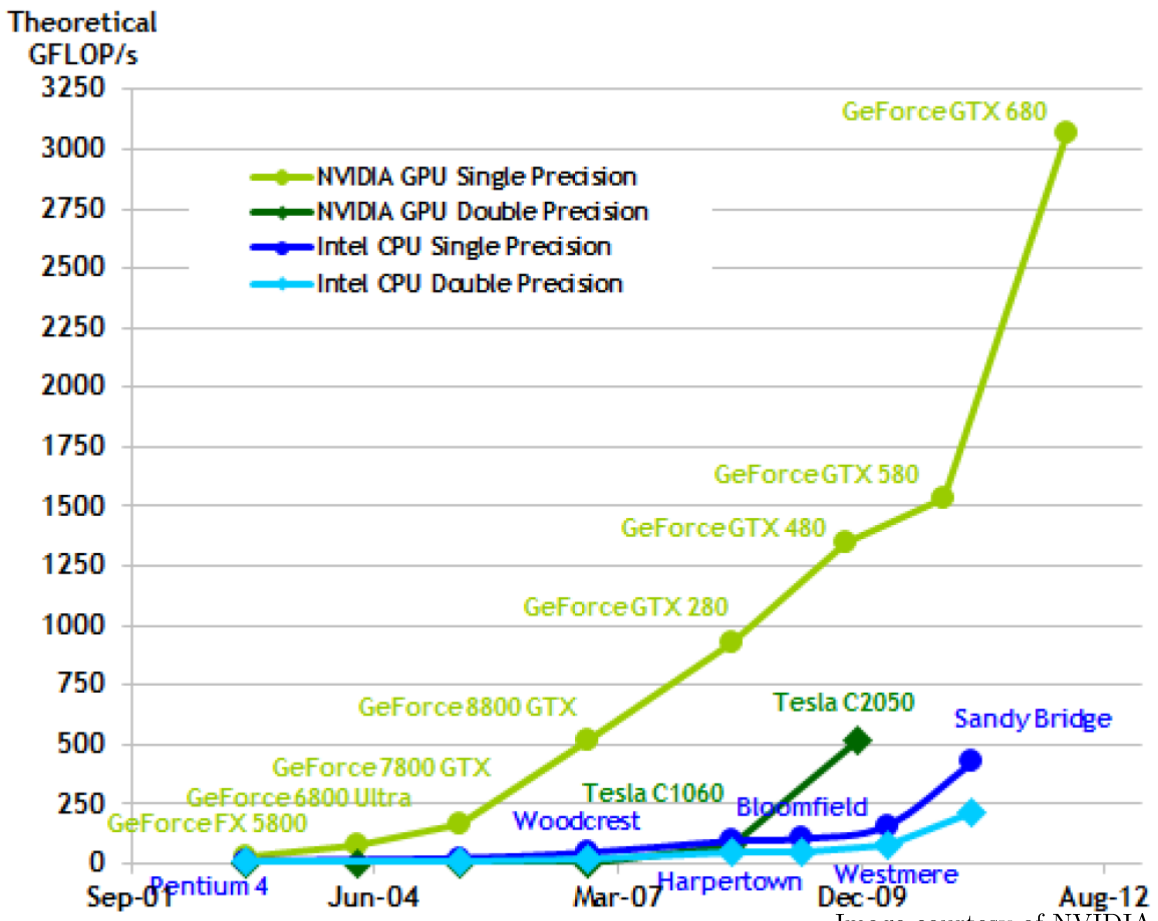


Image courtesy of NVIDIA

Figure 4.1 Peak computational performance of NVIDIA GPUs and Intel CPUs. [8]

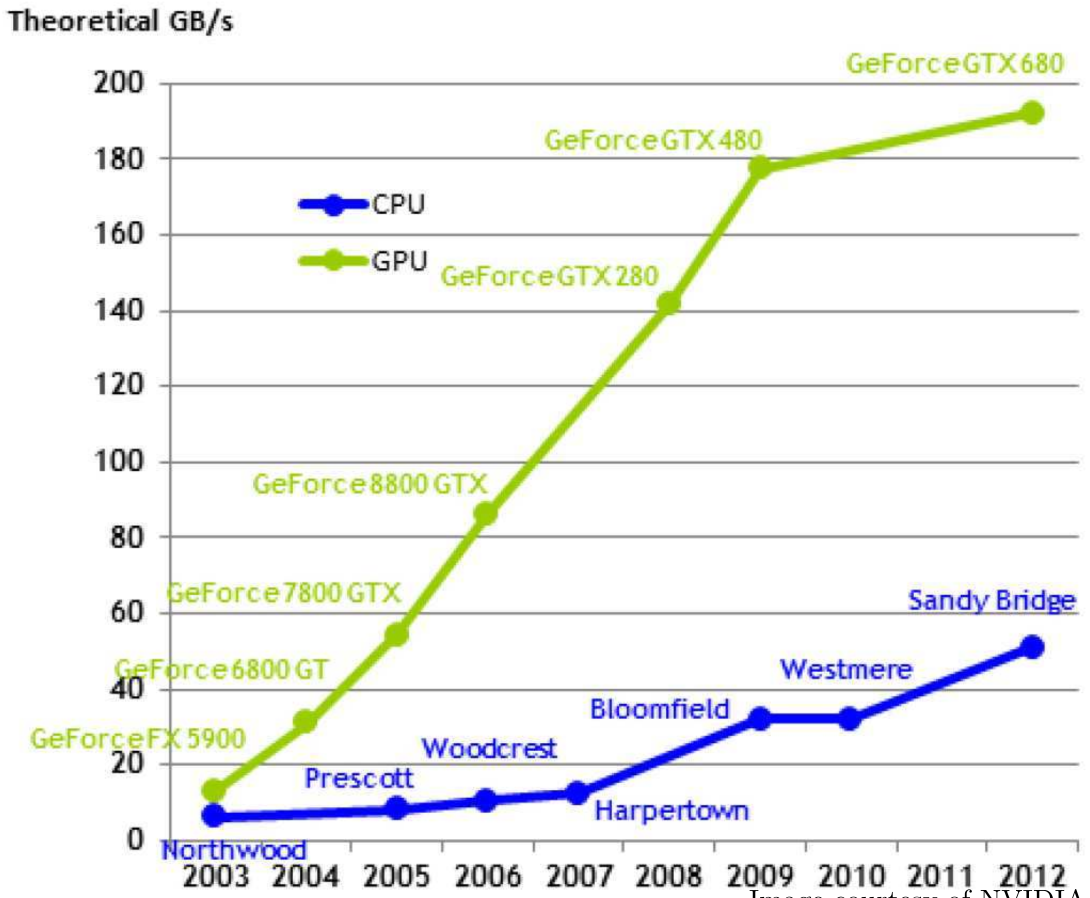


Image courtesy of NVIDIA

Figure 4.2 Peak memory bandwidth of NVIDIA GPUs and Intel CPUs. [8]

4.1 General CUDA Programming

This section focuses on understanding GPU hardware and the utilization of CUDA. However, since all current GPGPU frameworks share similar architectures, many of the higher level concepts can be used with other GPGPU frameworks.

4.1.1 GPU Hardware

In order to write efficient programs in CUDA, knowledge of the hardware's capabilities and limitations is required. This is mainly due to the massively parallel nature of GPU programs and the high level of control over memory management and kernel execution. On the other hand, knowledge of CPU architecture isn't necessary to write efficient programs, due to the optimizations modern compilers can automatically apply. The graphics card used in this research is NVIDIA's GTX 570, however, the general principles can be applied to other GPU's.

The GTX 570 GPU consists of 15 Streaming Multiprocessors (SMs), each SM contains 32 thread processors. This yields a total of 480 thread processors or "CUDA cores". The graphics card memory consists of two main types: shared memory and global "device" memory. The GTX 570 has 64 kB of shared memory per SM and 1.5 GB of device memory. This type of architecture is used with programs written using a Single Instruction Multiple Thread (SIMT) model. In the past, massively parallel vector computers used the Single Instruction Multiple Data (SIMD) which is similar to the SIMT model used by CUDA.

4.1.2 CUDA Software

In order to illustrate CUDA programming and the use of kernel functions, a simple vector addition function will be presented. Kernel functions are executed on the GPU concurrently by the GPU's multiprocessors The multiprocessors then utilize their thread processors to execute the required computation. Since most CUDA programs are written using CUDA C, we will start with a traditional serial CPU code for adding two vectors:

```
int main()
{
   ...

   for (int i=0;i < n;++i)
   {
      C[i] = A[i] + B[i];
   }
}
```

Listing 4.1 Serial vector addition.

Before proceeding, we must define some common CUDA terminology. CUDA kernel calls are made up of threads, blocks and grids. Threads are the individual processes that are executed within each SM, while blocks are a 1, 2 or 3-dimensional group of threads. A grid is a 1 or 2-dimensional set of blocks, each of which is executed by an SM (Figure 4.3). Since CUDA uses SIMT, each block will execute the same kernel when it is invoked. If a kernel is to be executed using a large array of data, each block and thread need to be aware of their position within the grid. These positions can be found using the intinsic functions blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y and threadIdx.z. Additionally the block and grid dimensions can be found using the following intrinsic functions: blockDim.x, blockDim.y, blockDim.z, gridDim.x and gridDim.y. Using the kernel concept and the intrinsic position functions, the CPU vector addition routine can be converted to a basic CUDA kernel:

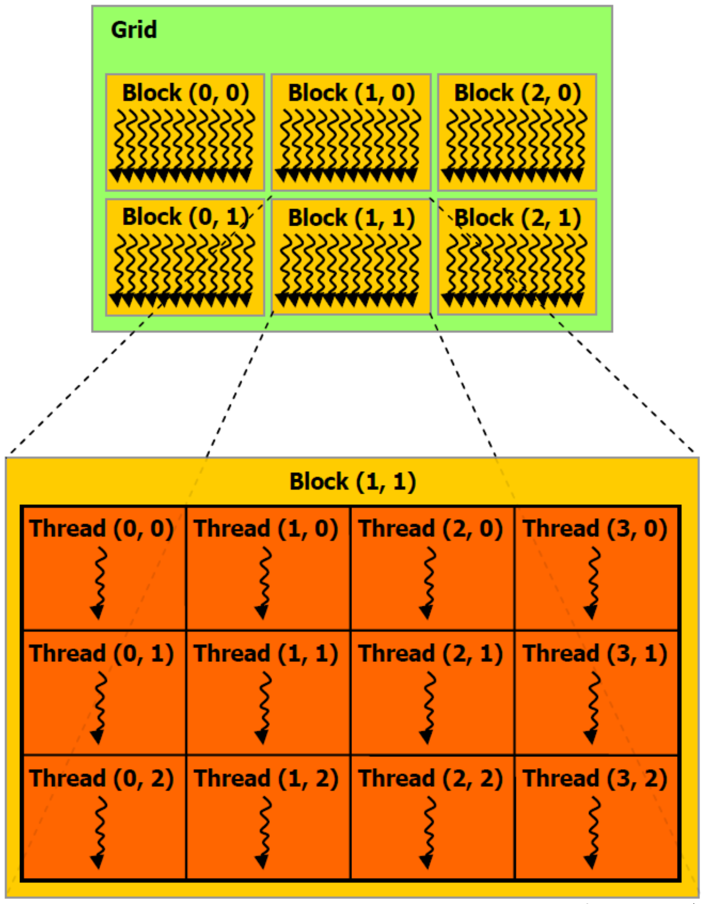


Image courtesy of NVIDIA

Figure 4.3 Block and Grid architecture for kernel execution. [8]

```
#define THREADS_PER_BLOCK 32

__global__
void vectorAdd(int n,double *A,double *B,double *C)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;

   C[i] = A[i] + B[i];
}

int main()
{
   ...

   int dimGrid = (n - 1) / THREADS_PER_BLOCK + 1;
   int dimBlock = THREADS_PER_BLOCK;

   vectorAdd<<<dimGrid,dimBlock>>>(n,A,B,C);
}
```

Listing 4.2   Simple parallel vector addition.

4.1.3 Memory Management

One key concept to the performance of CUDA programs is the proper usage of shared and device memory (Figure 4.4). In CUDA, the usage of the memory must be managed manually. Although the shared memory is small, the latency associated with accessing data stored here is significantly lower than the global memory (approximately 4 clock cycles vs. 400-600 clock cycles).

Due to this large discrepancy, shared memory needs to be used efficiently in order to achieve the full performance of the GPU. For peak performance, every kernel should typically follow this operational stencil:

1. Determine the grid location from the block and thread ID's.
2. Copy required data from device to shared memory.
3. Perform operations using only shared memory.
4. Copy results from the shared to the device memory.

The latency caused by memory copies between shared and device memory is typically largely hidden by the on-chip thread scheduler, as long as the memory is accessed sequentially. This is known as 'memory coalescing', various techniques to improve memory performance are given in detail in the CUDA Programming Guide [8]. By applying appropriate memory

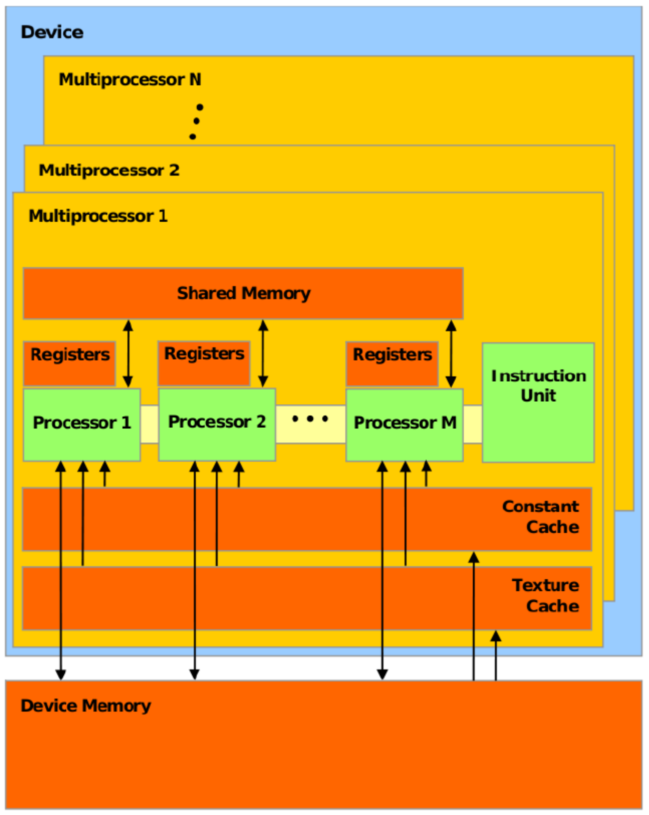


Image courtesy of NVIDIA

Figure 4.4 Hardware schematic of NVIDIA GPUs. [8]

management, we can significantly improve the performance of our previous CUDA kernel by replacing with the following:

```
#define THREADS_PER_BLOCK 32

__global__
void vectorAdd(int n,double *A,double *B,double *C)
{
   int j = threadIdx.x;
   int bDim = blockDim.x;
   int i = blockIdx.x * bDim + j;

   extern __shared__ char shared[];

   __syncthreads();

   double *A_s = (double *)shared;
   double *B_s = (double *)&A_s[bDim];
   double *C_s = (double *)&B_s[bDim];
   __syncthreads();

   A_s[j] = A[i];
   B_s[j] = B[i];

   __syncthreads();

   C_s[j] = A_s[j] + B_s[j];

   __syncthreads();

   C[i] = C_s[j];
}

int main()
{
   ...

   int dimGrid = (n - 1) / THREADS_PER_BLOCK + 1;
   int dimBlock = THREADS_PER_BLOCK;
   int smemSize = 3* THREADS_PER_BLOCK * sizeof(double);

   vectorAdd<<<dimGrid,dimBlock,smemSize>>>(n,A,B,C);
}
```

Listing 4.3 Parallel vector addition with shared memory.

It should be noted that the kernel call now has a third term ‘smemSize’ which tells the GPU how much shared memory is required by the kernel.

4.2 Solver Implementation

4.2.1 Runge-Kutta SIMPLER

The primary advantage of the Runge-Kutta SIMPLER algorithm over the Crank-Nicolson or Fully Implicit algorithms is it’s ease of parallelization. Since the algorithm is explicit in both space and time, threads can run independently of each other for a majority of the time step. This allows the on-chip thread scheduler to manage the thread processors and ensure none are idle. There are only a few key locations where thread synchronization must occur. These locations, such as setting up the pressure equation, need calculated data from surrounding nodes. Additionally, the Gauss-Seidel iterative process (a serial process) is replaced by the Runge-Kutta scheme which can be easily parallelized. This eliminates the need for complex parallel matrix solvers and further improves the solver's performance. The main limiter on performance is therefore efficient use of the shared and device memory space. This is complicated by the use of unstructured grids. As seen in the previous section, the shared memory allocated to each block needs to be declared during the kernel call. Allocating the shared memory for a structured grid is simple since there are always a known number of element or node values needed for calculations. When using an unstructured grid, this is complicated by the varying number of neighboring nodes and elements for each face. For this research, the maximum number of connected elements and nodes is first determined from the grid data, denoted 'N_{max}'. Since each block will be allocated the same amount of shared memory, the max number was then used to allocate shared memory space to ensure each kernel will have enough storage (Listing 4.4). Careful usage of the shared memory is required though, since the maximum number of double precision numbers allowed per thread is only 256 on the GTX 570.

```
int smemSize = N_max * 4* sizeof(double);
```

Listing 4.4 Shared memory sizing

Another advantage of the explicit Runge-Kutta routine is scalability. The main limitation to GPU computing is the limited device memory space. This can be mitigated using the fully explicit routine, since the data necessary for each thread is at most the values at the desired node and the surrounding nodal values. Additionally, with the elimination of the Gauss-Seidel iterations, there is no time in the code when values for the entire grid are needed at one time. For large grids (node points $> 350,000$), the device memory will no longer be able to hold all of the data needed for the entire computational domain. One solution is to split the domain into smaller sub-domains and compute each sub-domain in stages during each time step. After each sub-domain is computed the device memory is swapped to contain the relative data points. This allows the maximum grid size to be limited only by the maximum system memory, typically 8-20 times more than the device memory, at the expense of latency due to memory transfers from the host to the GPU. This latency can be further reduced to a degree by using asynchronous data transfers and kernel executions. Asynchronous executions allows the GPU to process a kernel while transferring data to and from the host limiting and sometimes completely masking the latency caused by memory transfers between the hardware. Additionally, the above process can be adapted to extend the algorithm to use multiple GPUs and increase computational performance. Since the two scaling options are closely related, it is simple to implement one or both of these options once a general procedure is formulated. Detailed example of multi-GPU computing can be found in the NVIDIA CUDA C Programming Guide [8].

4.2.2 Fortran-CUDA interoperability

In terms of speed, it is generally preferable to perform all of the computation on the GPU, however, it is often necessary to use legacy code with CUDA kernels. This interoperability can be used to accelerate a few time intensive subroutines in an existing CPU code or allow a primarily GPU based code to use CPU functions that cannot be converted to CUDA or would not be efficient to execute on the GPU. In this research, the code uses a Fortran based unsteady rotor model developed by Guntupalli [4] along with the CUDA based solver. The general outline for calling CPU functions in a GPU code is:

1. Copy data needed by the CPU function(s) from GPU to host.
2. Execute CPU function(s).
3. Copy calculated data from host to GPU

It should be noted that there is no need to copy the original data back to the card if it has not been manipulated by the CPU code. A similar process can be used for calling GPU functions from CPU code:

1. Copy data needed by the GPU kernel(s) from host to GPU.
2. Execute GPU kernel(s)
3. Copy calculated data from GPU to host.

However, standard Fortran implementations do not have the intrinsic functions to interact with the GPU, so a wrapper function must be created. The wrapper function takes advantage of Fortran's ability to call C functions natively and C's capability to transfer memory to the GPU and execute CUDA kernels. Since C and Fortran can be mixed natively, the only overhead going from Fortran to CUDA is the latency cause by transferring data from host to GPU. Referring back to our original vector addition example, Listings 4.5 and 4.6 demonstrate how a CUDA kernel can be wrapped in a C function so it can be called from Fortran. This type of interoperability is required for any useful CUDA program since CUDA is a language that can only handle computation. This means data input/output and any other non computational activities have to be done by CPU code of some type. In this research, Fortran-Cuda interoperability is used primarily for reading the input data, calculating the rotor source terms, and writing result files.

```
#define THREADS_PER_BLOCK 32

__global__
void vectorAdd(int n,double *A,double *B,double *C)
{
   int j = threadIdx.x;
   int bDim = blockDim.x;
   int i = blockIdx.x * bDim + j;

   extern __shared__ char shared[];

   __syncthreads();

   double *A_s = (double *)shared;
   double *B_s = (double *)&A_s[bDim];
   double *C_s = (double *)&B_s[bDim];

   __syncthreads();

   A_s[j] = A[i];
   B_s[j] = B[i];

   __syncthreads();

   C_s[j] = A_s[j] + B_s[j];

   __syncthreads();
   C[i] = C_s[j];
}

extern "C"
{
void addvectors_(int *N,double *A,double *B,double *C)
{
   ...

   int dimGrid = (*N - 1) / THREADS_PER_BLOCK + 1;
   int dimBlock = THREADS_PER_BLOCK;
   int smemSize = 3* THREADS_PER_BLOCK * sizeof(double);

   vectorAdd<<<dimGrid,dimBlock,smemSize>>>(*N,A_d,B_d,C_d);
}
}
```

Listing 4.5 Parallel vector addition kernel with Fortran wrapper.

```
PROGRAM main
IMPLICIT NONE

INTEGER, PARAMETER :: N = 1000
REAL*8 :: A(N), B(N), C(N)

...
CALL addVectors(N,A,B,C)
...

END PROGRAM main
```

Listing 4.6 Fortran call to CUDA function.

4.2.3 CUDA implementation specifics

For this research, all of the calculations were done on the GPU with the exception of the Gauss-Seidel iterations used to solve for pressure. The Gauss-Seidel iteration method not only updates the i^{th} pressure of the loop but also the pressure coefficients of the neighboring nodes. As a result of this, splitting the Gauss-Seidel iterations by using a coloring scheme or other popular methods becomes very difficult since each region has to be able to update the center node and all of the neighbor nodes without experiencing a race condition. However, leaving it on the CPU was deemed acceptable since the computation time used to solve for the pressure is relatively low in comparison to the other computations (approximately 4%), the computation time percentages for a typical run are listed in Appendix D. As a result of this, the maximum performance gain is limited by Amdahl's Law:

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}} \tag{4.1}$$

where $S(N)$ is the total speed-up due to the speed-up of the parallel sections, P is the percentage of the code (in computation time) that is in parallel, and N is the speed-up of the parallel sections. Taking the limit of this equation where $N \to \infty$ yields a maximum speed-up of 24.39 times. The CUDA implementation is therefore bound by this limit and the limit can be used to measure the performance of the CUDA code vs. the idealized case. It should be noted that the speed-ups listed in the Chapter 5 are relative to the Crank-Nicolson solver and can be greater 24.39 times since the Runge-Kutta algorithm itself has a performance gain and is then enhanced by CUDA.

CHAPTER 5. RESULTS

5.1 3D Unstructured Rung-Kutta Code Validation

The Runge-Kutta 3D unstructured flow solver was validated using benchmark problems with and without GPU acceleration. Each benchmark problem and the results obtained are described in the following sections.

5.1.1 Lid Driven Cavity

The standard test case for both two-dimensional and three-dimensional solvers is the lid driven cavity. In this case, the cavity is a cube with edge lengths of one ($L = 1$) and a top surface ($z = 1$) moving at a constant velocity of $1m/s$ (Figure 5.1). The cavity walls are all treated as no-slip boundary condition. The moving lid drives a circulation inside the cavity due to the transport of shear stress. The Reynolds number for the lid driven cavity can be calculated using the equation:

$$Re = \frac{\rho U_{lid} L}{\mu} \tag{5.1}$$

where ρ, μ and U_{lid} are the fluid density, viscosity and lid velocity, respectively. Reynolds numbers of 100, 400 and 1000 were simulated and compared against the Fully-Implicit solution. The grid used for is a $33x33x33$ stretched grid consisting of 196,608 tetrahedral elements. A stretch grid was used in order to capture the velocity profile near the walls more accurately. The time step and computation time for the various solvers can be seen in Table 5.1.

By comparing the centerline velocity profiles for $Re = 100$ between the traditional Crank-Nicolson solver and the Runge-Kutta, it can be seen that the solutions show an

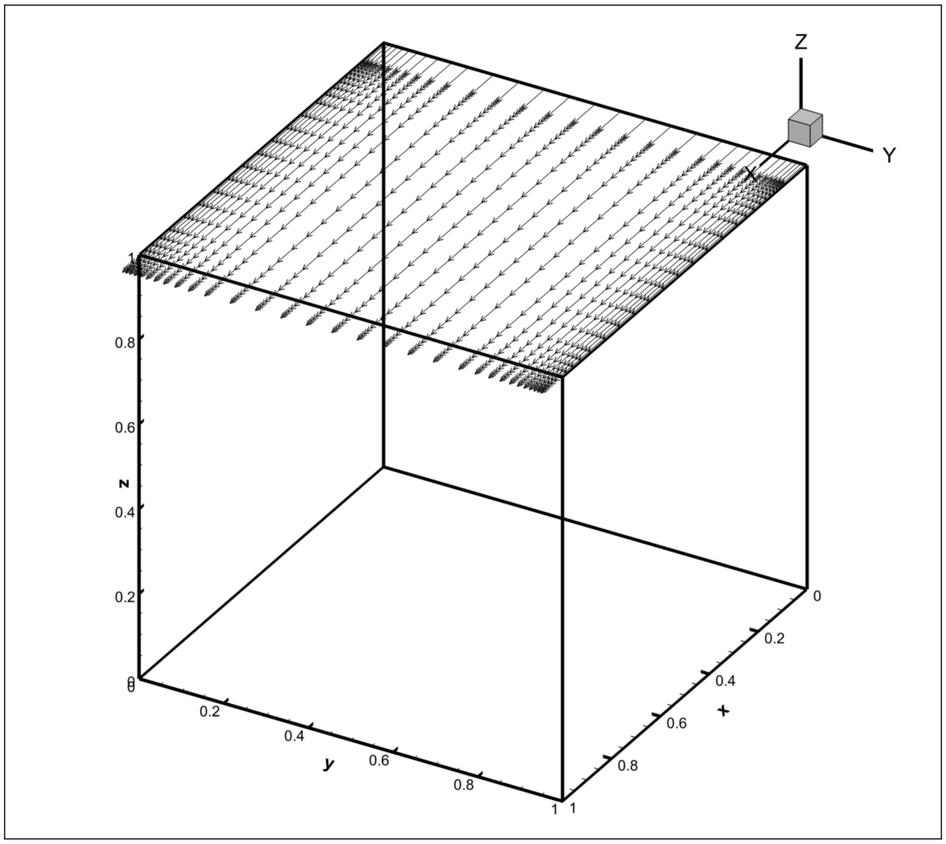
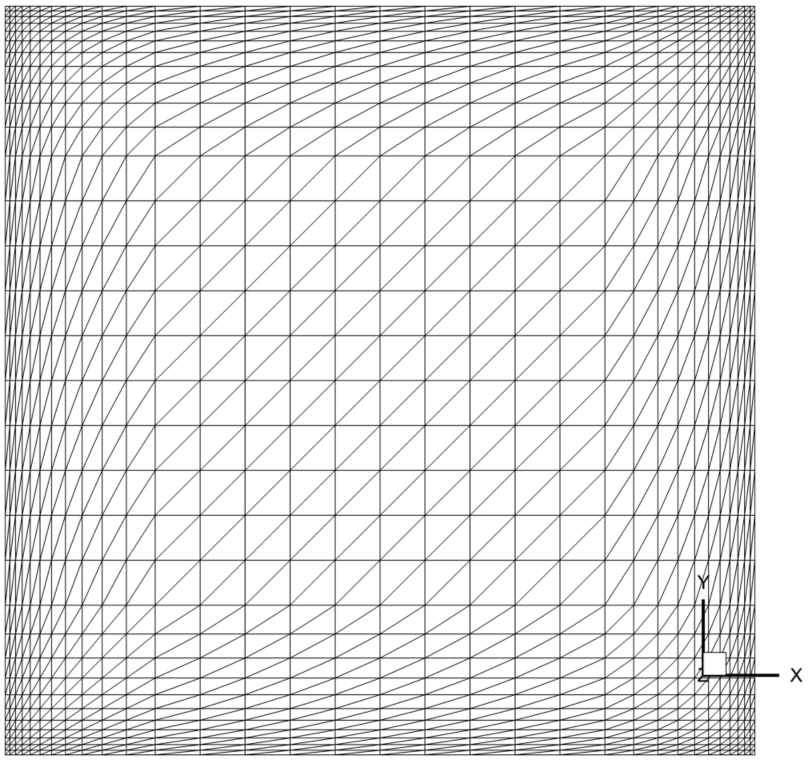


Figure 5.1 Schematic of lid driven cavity

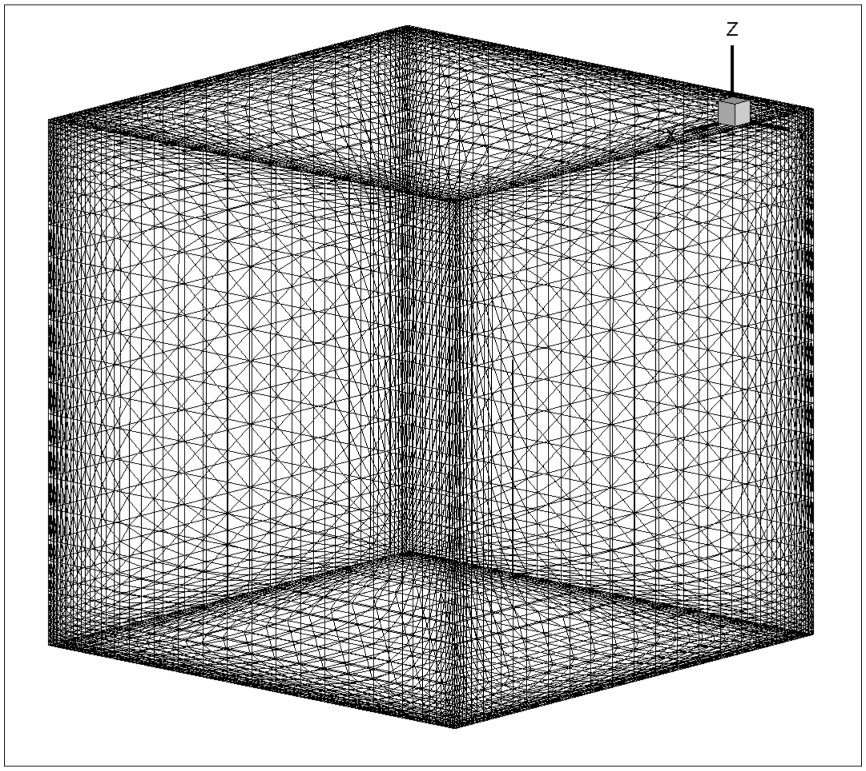
overall good agreement (Figure 5.3). Further, the velocity vector plots at the mid-planes of the computational domain show the correct physical phenomenon. Figure 5.6 shows the primary vortex centered slightly above the center of the cavity, while, Figure 5.5 shows two counter rotating vortices's that move toward the corners of the $y-z$ plane. The results for Reynolds numbers of 400 and 1000 also show excellent overall agreement between the different time integration schemes. From Figure 5.10 it can be seen that the solution reached convergence to a satisfactory extent. The explicit Runge-Kutta scheme requires a smaller time step than the Crank-Nicolson or Fully Implicit solution in order to maintain stability. It was found through experimentation that a time step one order of magnitude smaller than used by the Crank-Nicolson solution yielded good stability over a broad variety of cases.

Algorithm	Time Step	Exec. time (s)	Speed-up
Crank-Nicolson	0.01	868803.217	1x
Runge-Kutta	0.001	201344.894	4.3x
Runge-Kutta in CUDA	0.001	10642.866	81.63x

Table 5.1 Performance comparison of integration methods for the lid driven cavity over a fixed simulation time

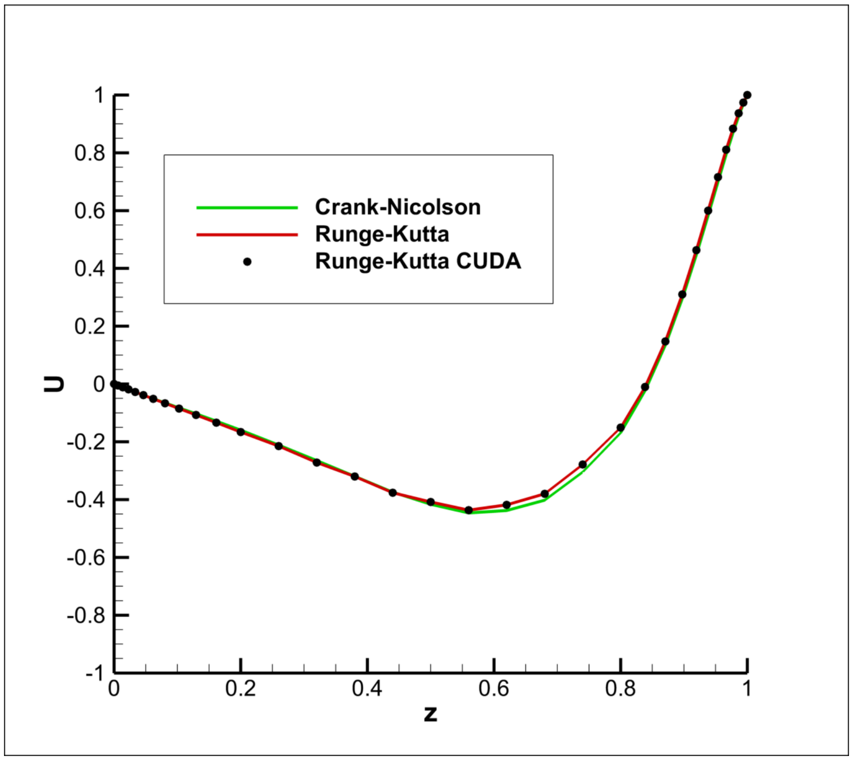


(a) 33 x 33 x 33 stretched grid, Number of nodes = 35937

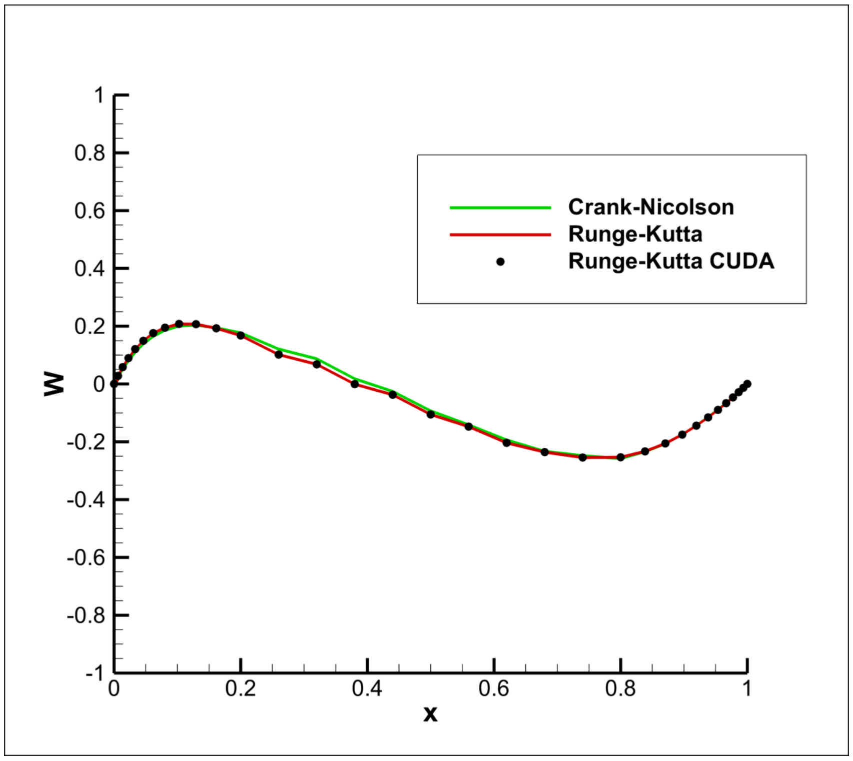


(b) 33 x 33 x 33 stretched grid, 3D

Figure 5.2 Computational Grids



(a) U-velocity profile



(b) W-velocity profile

Figure 5.3 Centerline velocity profiles, $Re = 100$

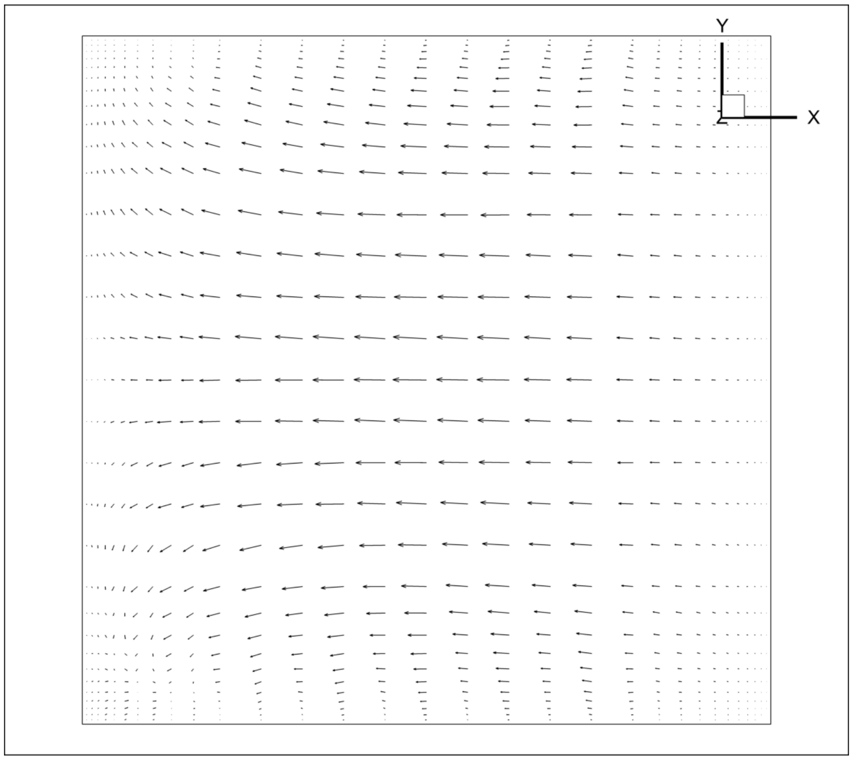


Figure 5.4 Velocity vectors for the X-Y plane, $z = 0.5$, $Re = 100$

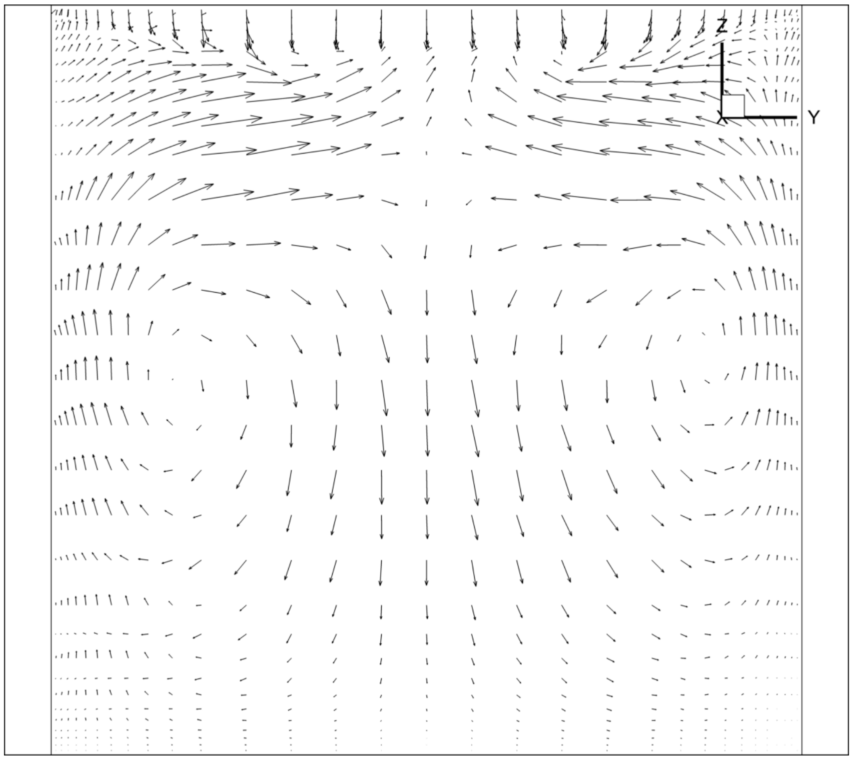


Figure 5.5 Velocity vectors for the Y-Z plane, $x = 0.5$, $Re = 100$

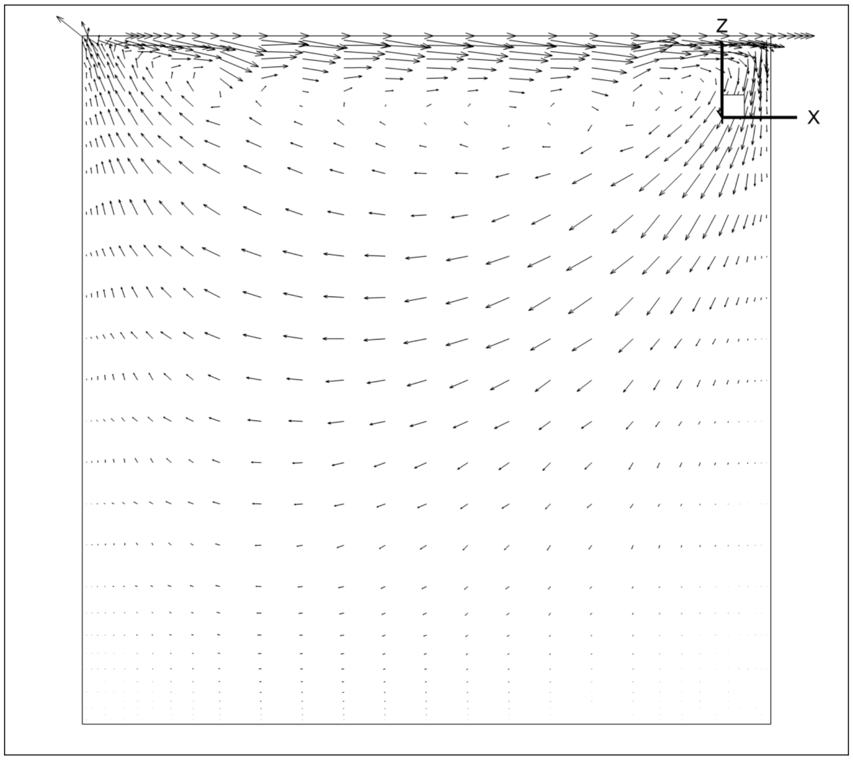


Figure 5.6 Velocity vectors for the X-Z plane, $y = 0.5$, $Re = 100$

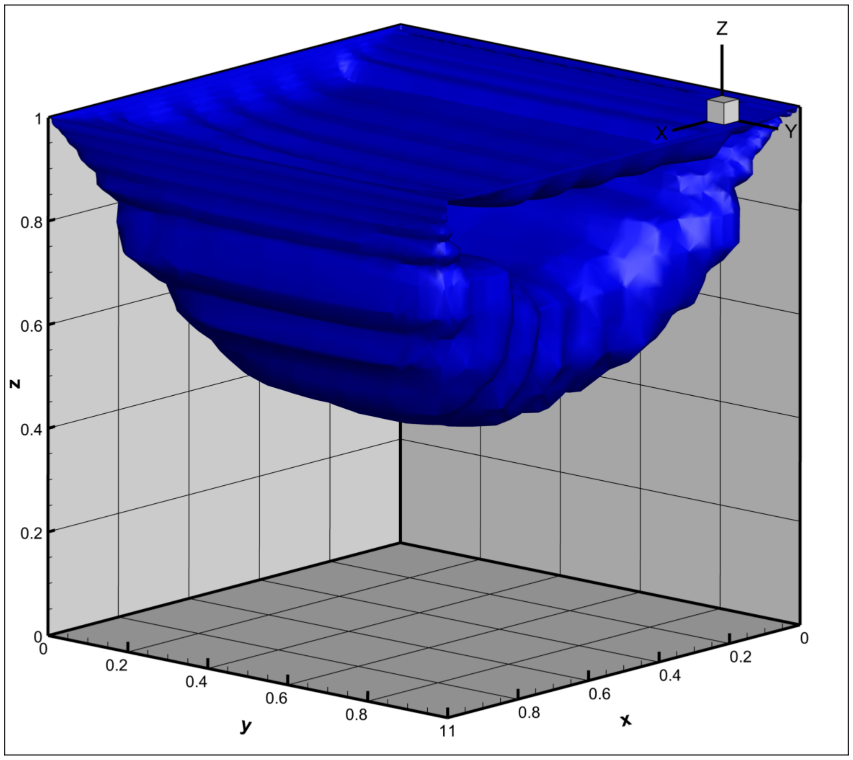
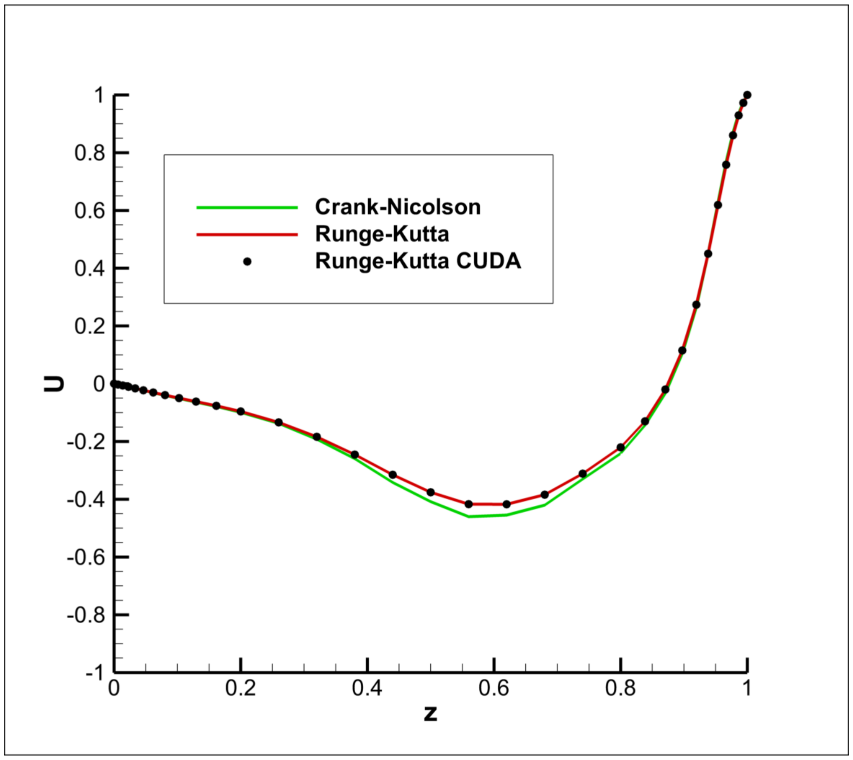
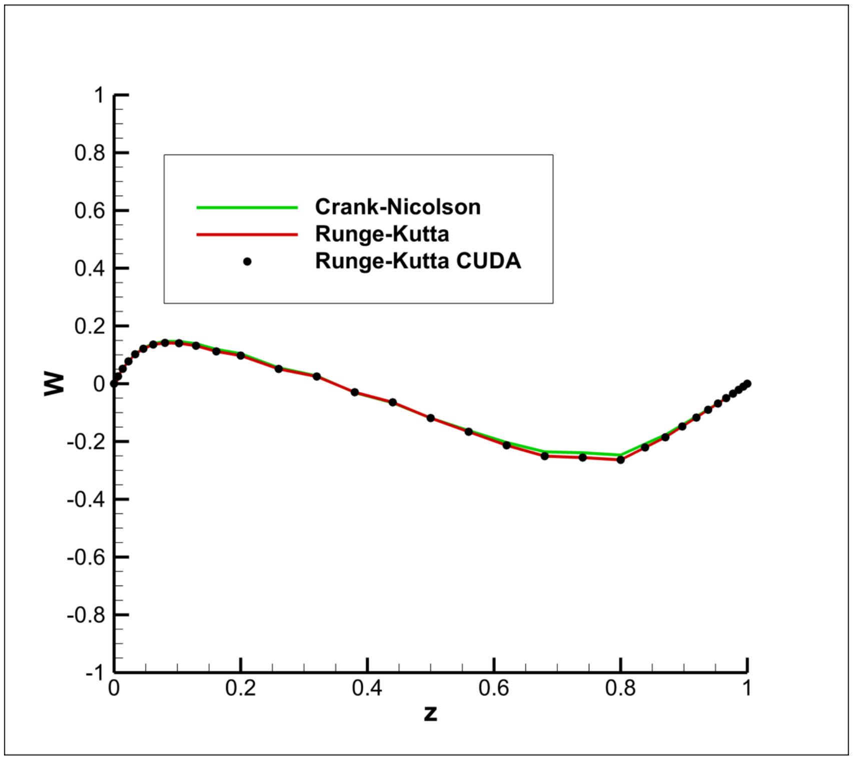


Figure 5.7 Isosurface of velocity magnitude, $Re = 100$

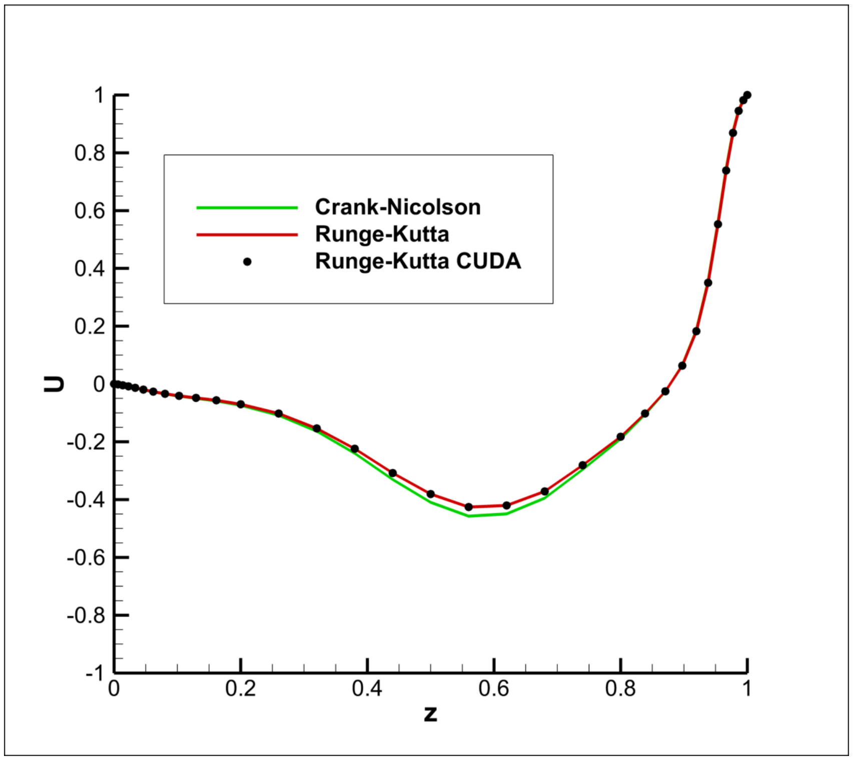


(a) U-velocity profile

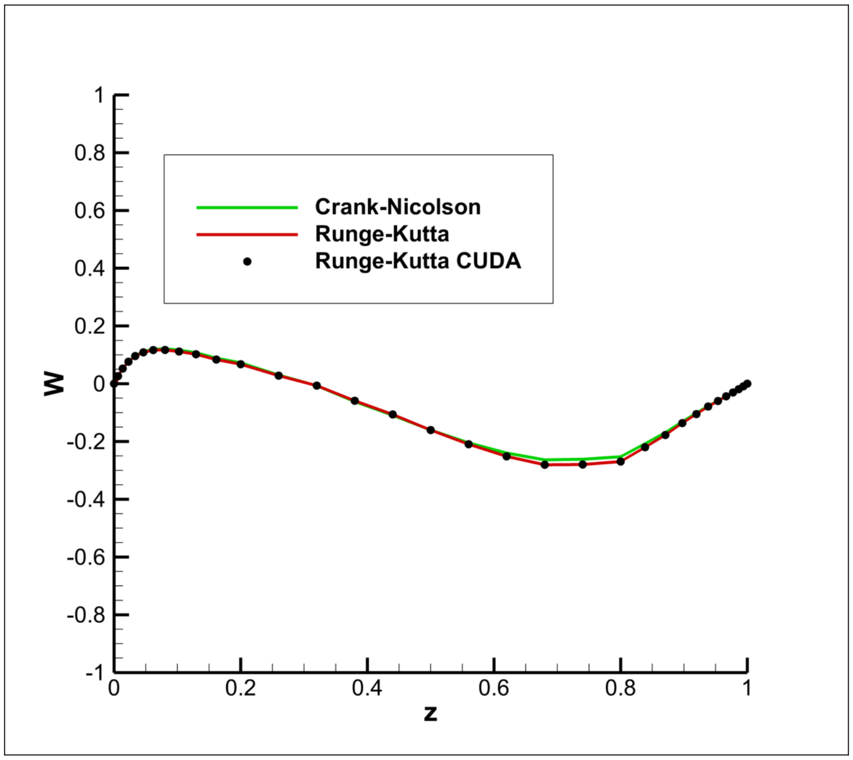


(b) W-velocity profile

Figure 5.8 Centerline velocity profiles, $Re = 400$



(a) U-velocity profile



(b) W-velocity profile

Figure 5.9 Centerline velocity profiles, $Re = 1000$

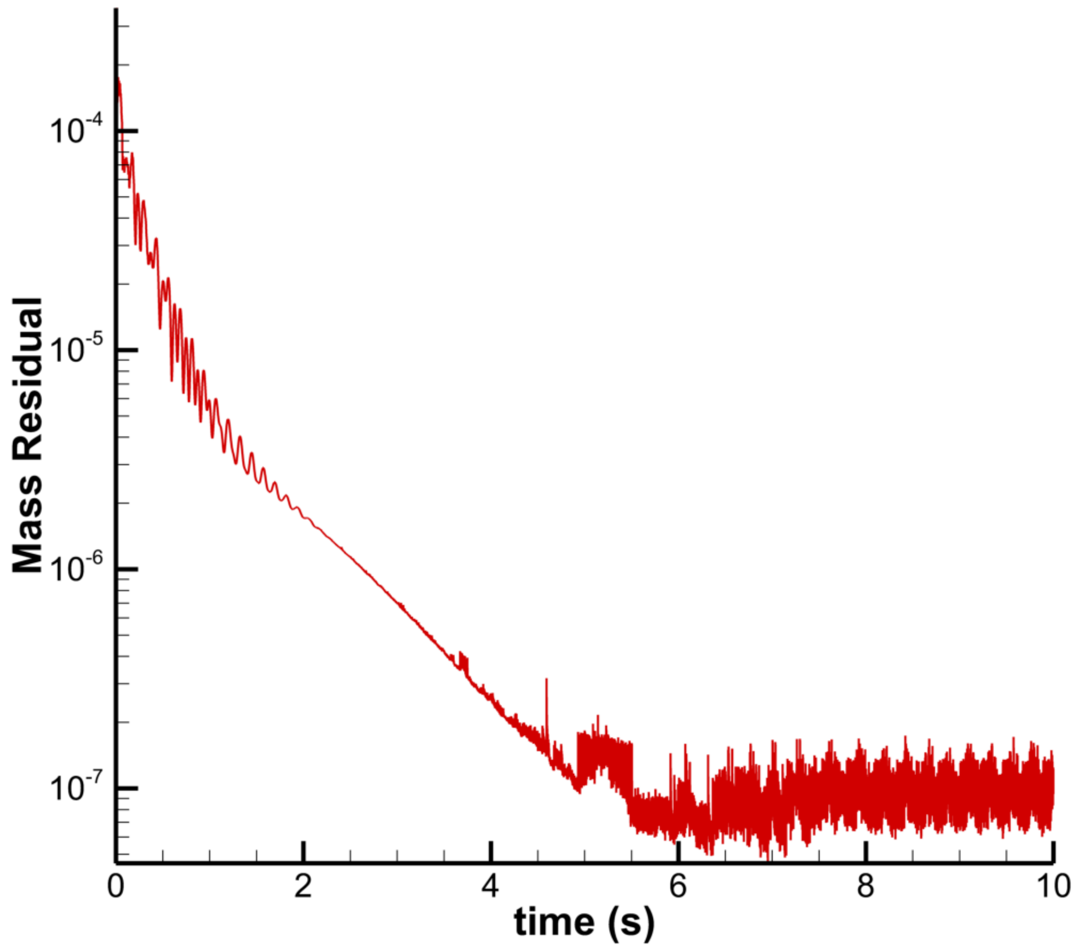


Figure 5.10   Mass residual of Runge-Kutta solver for lid driven cavity

5.1.2 Unsteady Rotor

In order to test the Runge-Kutta algorithm with the unsteady rotor, the Rabbot rotor is simulated at a collective pitch of 4.5°. The grid used for the rotor simulation has 9261 nodes and is shown in Figure 5.11. One important parameter of the rotor's performance while hovering is the coefficient of thrust C_T. In Figure 5.12, the thrust coefficient versus time is shown for the various solvers tested. The Crank-Nicolson unstructured solver with an unsteady rotor produces valid results [4]; this solver is used as the baseline comparison for the Runge-Kutta solver. From Figures 5.12 and 5.13, it is clear that the time history of the thrust coefficients are comparable to those of the Crank-Nicolson solver. Experimental data is available, against which the solution can be compared. To generate a more accurate comparison, the C_T of the steadily oscillating portion is averaged over 50 rotor rotations. These averaged values along with the execution times for each solver are shown in Table 5.2. The average thrust coefficient from each algorithm shows excellent congruency with the experimental data, producing a difference of less than 2%.

Algorithm	Time Step	C_T	Exec. time (s)	Speed-up
Experimental	-	0.00217	-	-
Crank-Nicolson	0.0025	0.00221	170454.544	1x
Runge-Kutta	0.00025	0.002188	37055.336	4.6x
Runge-Kutta in CUDA	0.00025	0.002188	2035.976	83.72x

Table 5.2 Performance comparison of integration methods for the unsteady rotor over a fixed simulation time

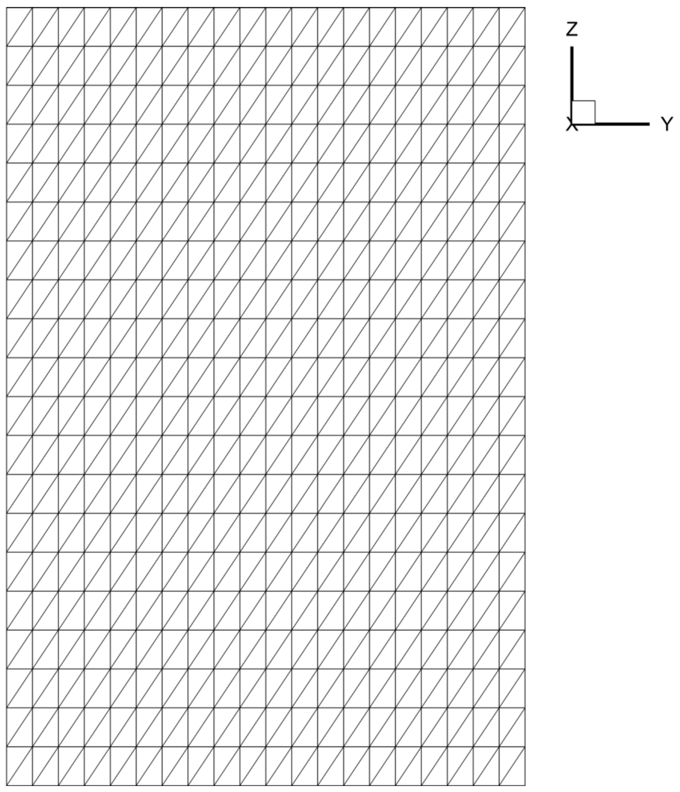


Figure 5.11 Unsteady rotor grid with 9261 nodes.

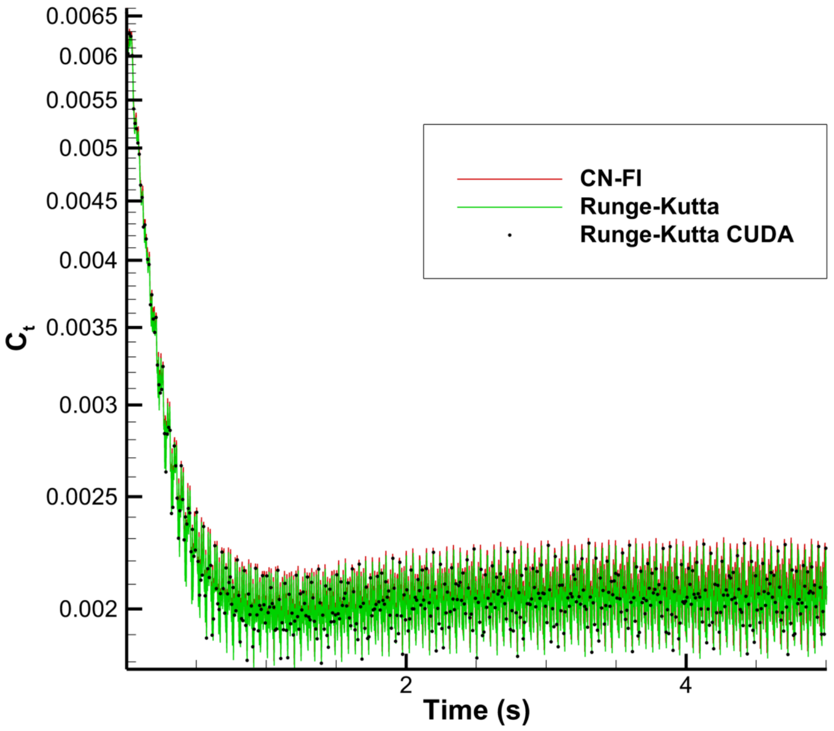


Figure 5.12 Thrust coefficient C_T vs. time for Crank-Nicolson, Runge-Kutta and Runge-Kutta in CUDA.

In order to maintain stability the explicit routine need to run a time step approximately one order of magnitude smaller than the Crank-Nicolson or Fully Implicit solver to maintain stability. This means that for the same computation time range, the explicit codes need to run ten times as many time steps as the Crank-Nicolson or Fully Implicit solver need. In spite of this, the CPU Runge-Kutta solver is still faster than the Crank-Nicolson or Full Implicit solver, primarily because of the elimination of the convergence sub-iterations and the elimination of the pressure correction equation. The speed-up is further increased by the CUDA implementation and the computation time is reduced from approximately 2 days to just under 34 minutes.

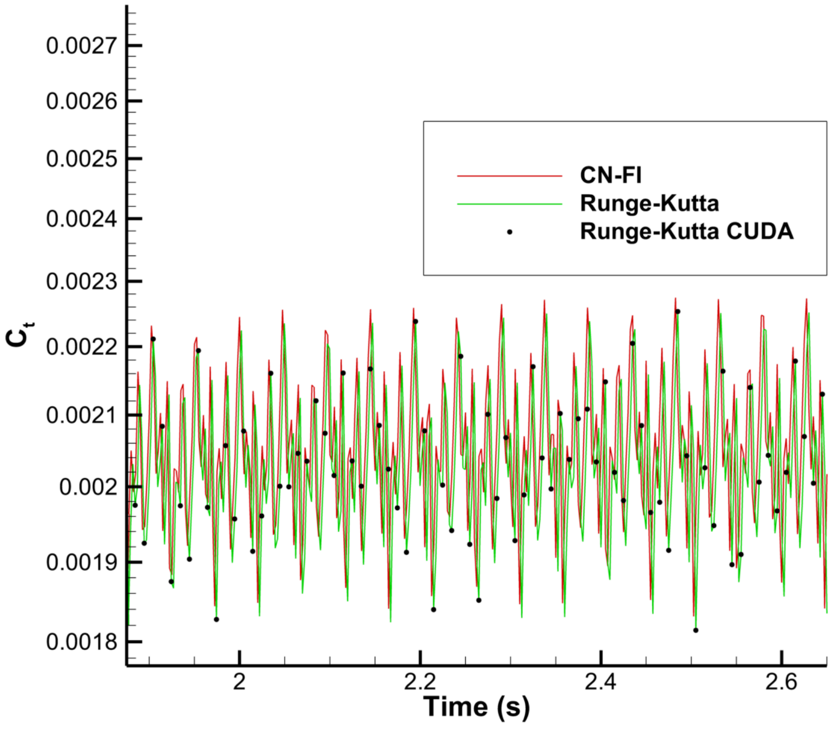


Figure 5.13 Thrust coefficient C_T detail.

Next, the rotor flow characteristics as calculated by the Runge-Kutta algorithm are presented. The instantaneous pressure contours above and below the rotor can be seen in Figure 5.1.2. This pressure differential between the top and bottom of the blades is the expected result, therefore, the physics of the flow are being maintained. Figure 5.14 shows velocity contours on a vertical plane slice through center of the rotor. The velocity field agrees well with previous results, showing a high velocity jet below the rotor and a large slow moving inflow. Finally, Figures 5.16 and 5.17 present the vorticity magnitude on the rotor plane and in the 3D wake, respectively.

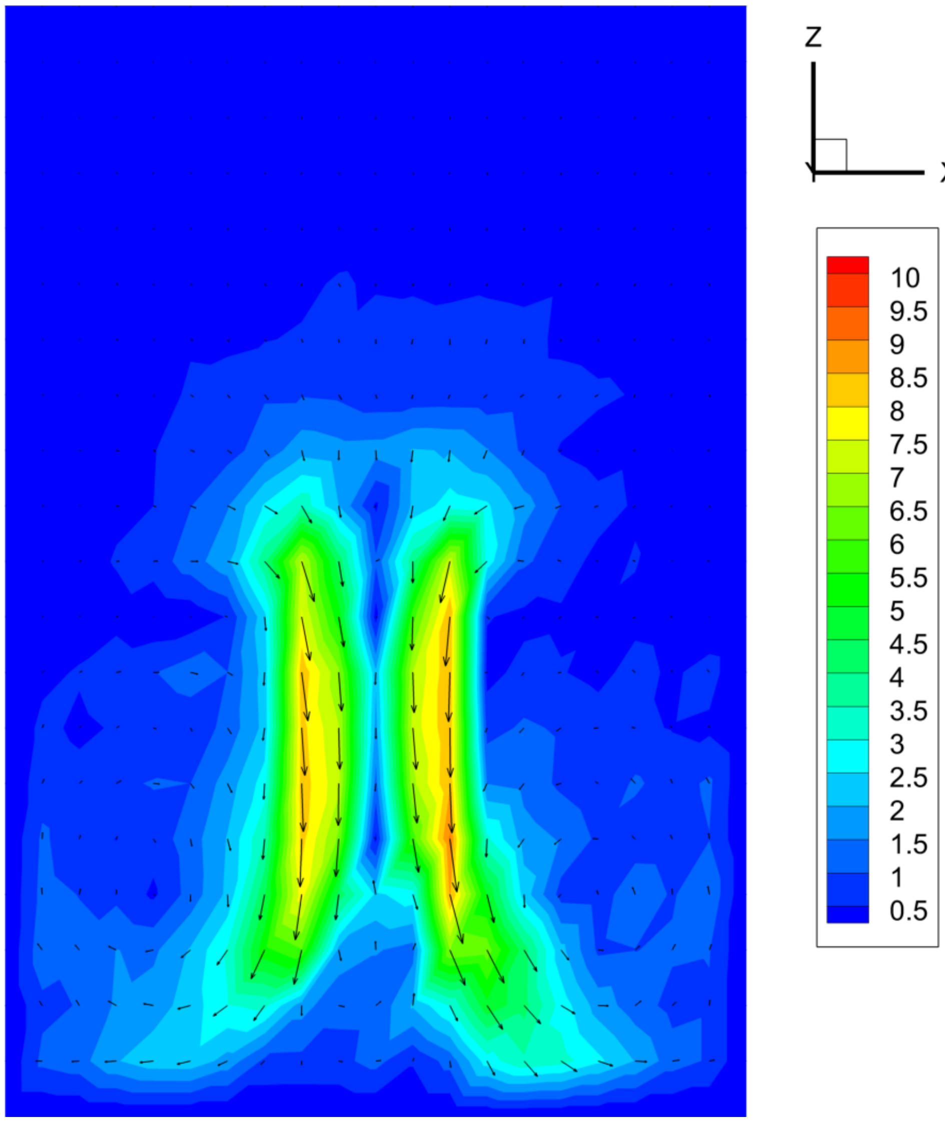
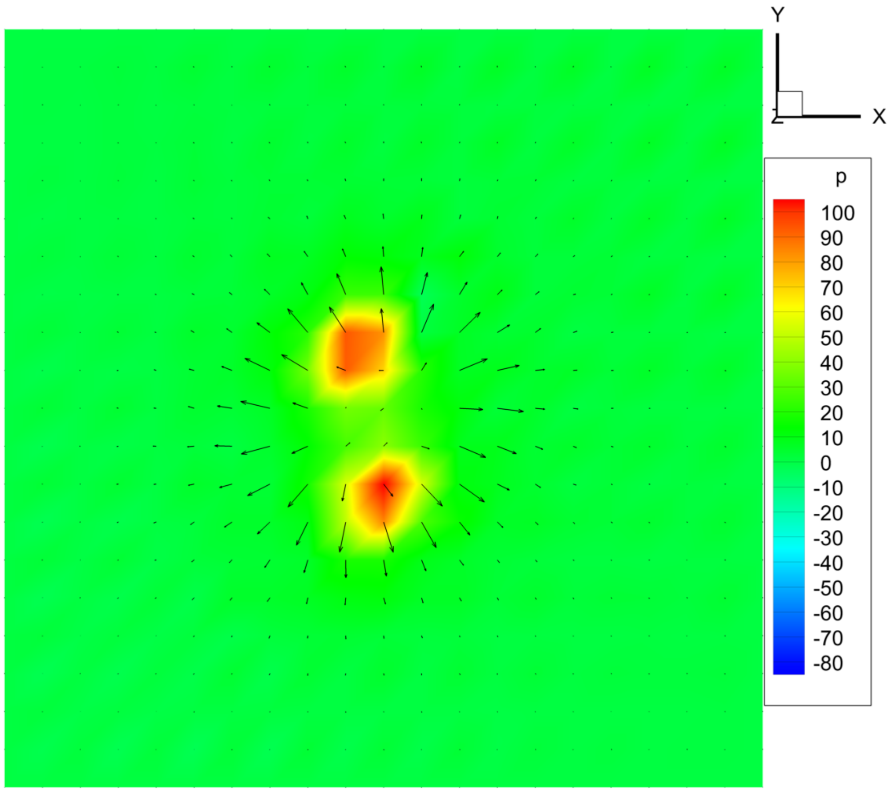
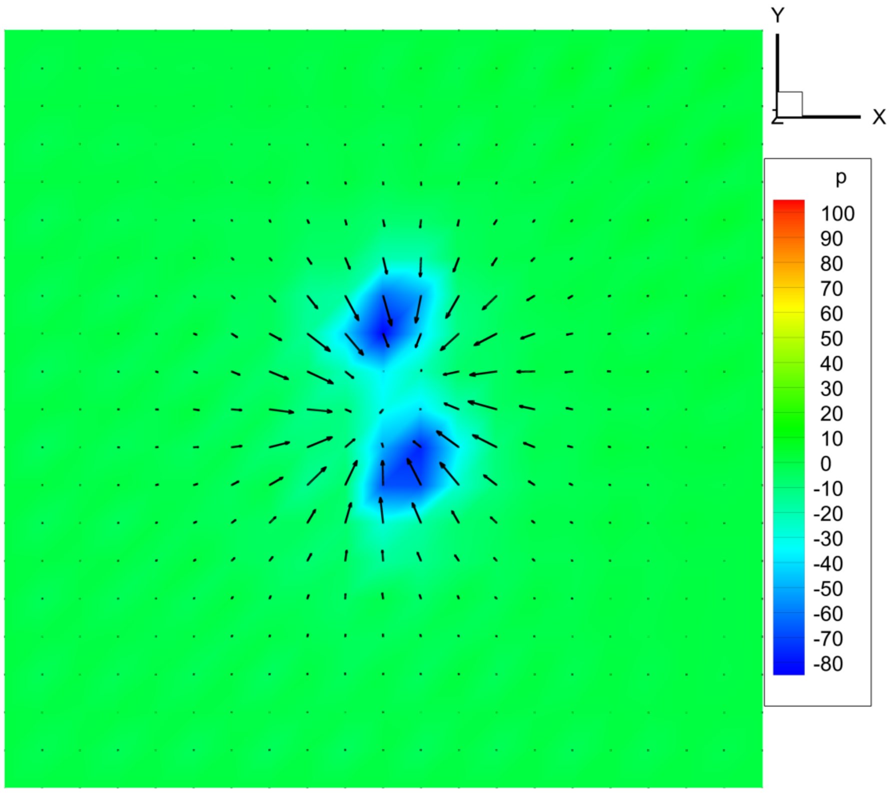


Figure 5.14 Velocity magnitude contours at the centerline, $t = 5s$.



(a) Below the rotor



(b) Above the rotor

Figure 5.15 Pressure contours above and below the rotor at $t = 0.125s$

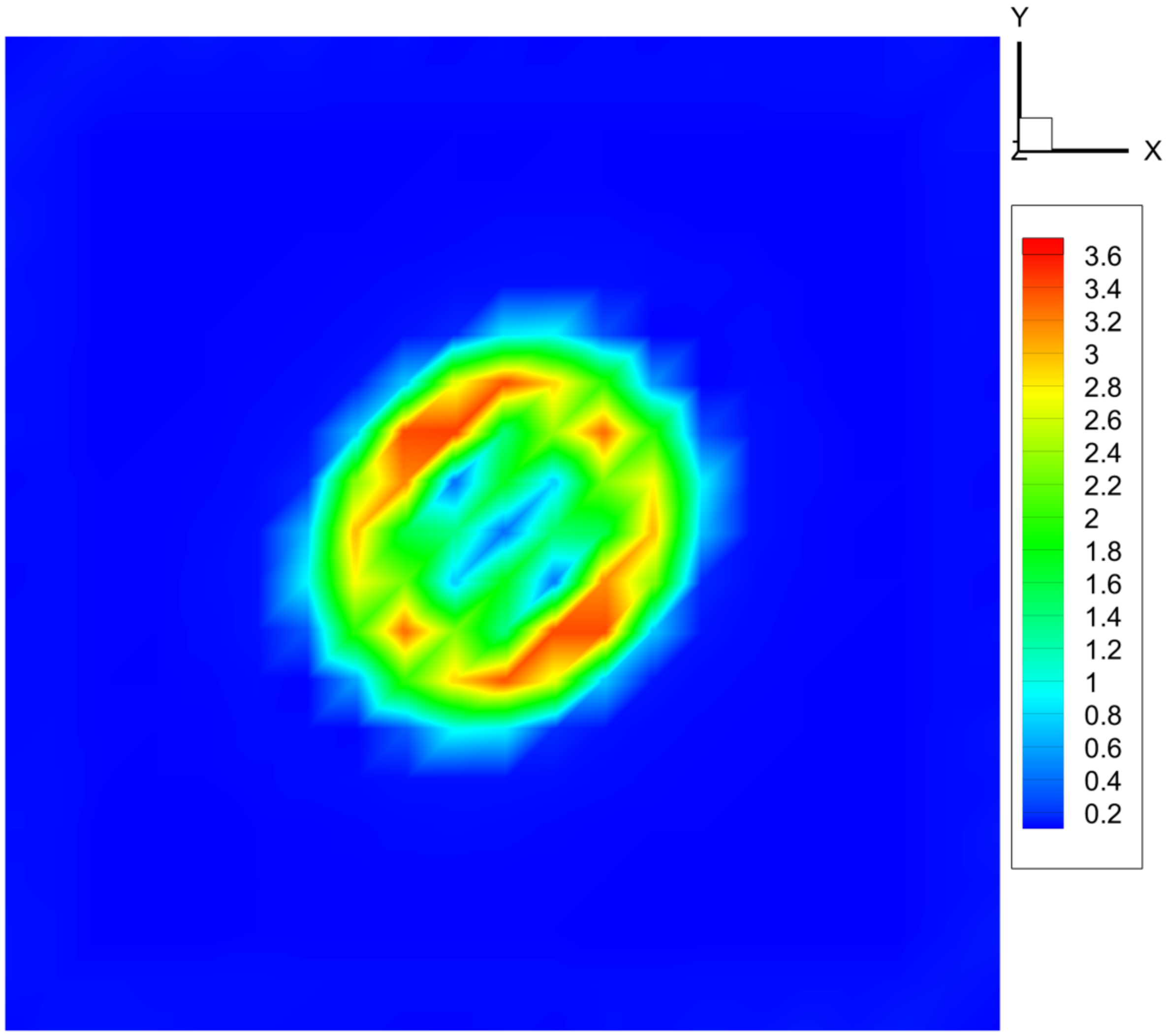


Figure 5.16 Vorticity magnitude at the rotor. $t = 5s$.

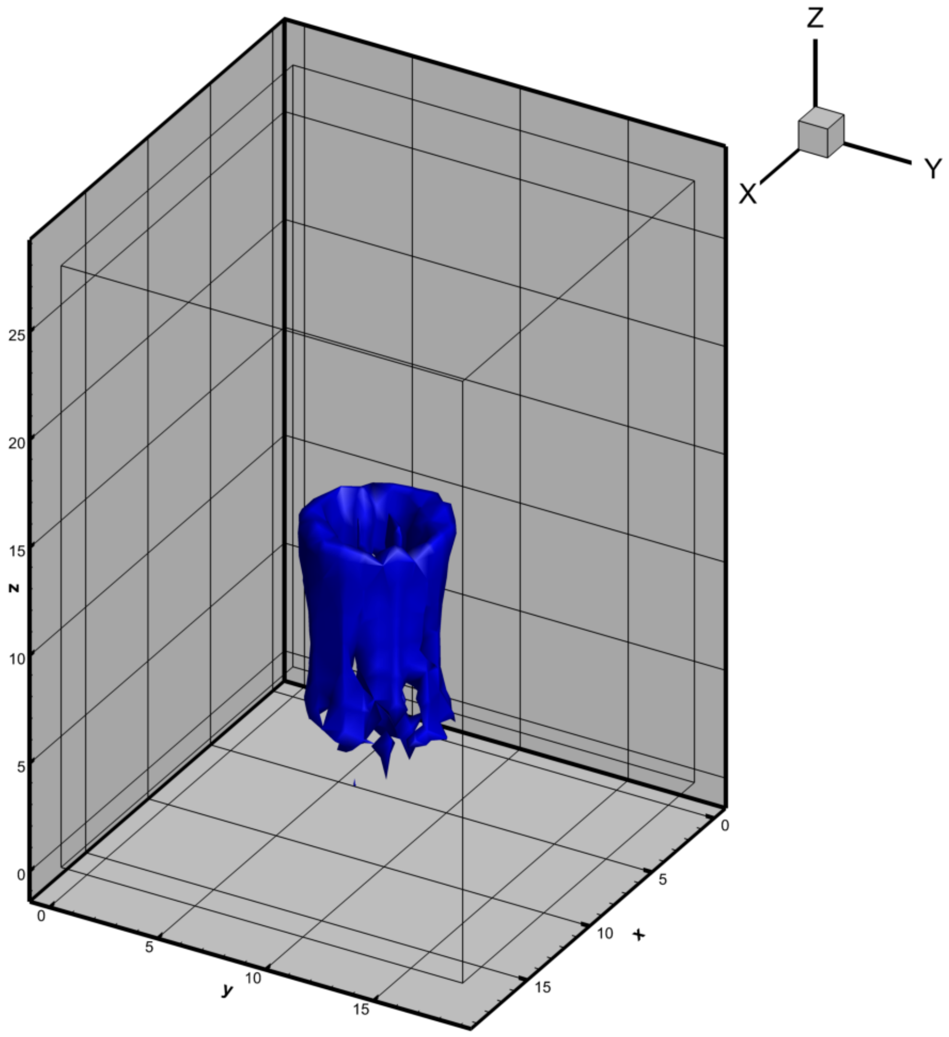


Figure 5.17 Isosurface of vorticity magnitude, $t = 5s$.

CHAPTER 6. CONCLUSION AND RECOMMENDATIONS

In this research, an unstructured, time-accurate, explicit solver is presented for 3-D incompressible flows. The incompressible, unsteady Navier-Stokes equations were solved using median-dual based control volumes on tetrahedral grids. A vertex centered finite volume discretization was employed with an artificial velocity field to achieve pressure-velocity coupling. The discretized equations were solved using a SIMPLER based algorithm. Implicit and Runge-Kutta based explicit time integration schemes were developed and implemented. The resulting solver was validated using confined flow in a driven cavity and using an unsteady rotor in hover. A good agreement between the time integration methods and existing numerical data was observed. It was found that the explicit time integration is capable of capturing unsteady, three-dimensional effects with less computational effort. In addition, the explicit solver was implemented in a parallel environment utilizing the GPU. This allowed even lower computation times due to the parallel nature of the explicit algorithm.

Although the results show promise, further work is necessary to make the algorithm more general and robust. A comparison should be done between implicit Runge-Kutta, explicit Runge-Kutta and other high order explicit methods to obtain the differences and computation times. The explicit flow solver needs to be tested with more external flows and tested with rotor-body configurations. Extending the grid to a general arbitrary control volumes would also enhance the usefulness of the solver.

APPENDIX A. Shape Function for a General Variable Φ

Assuming an interpolation function of $\Phi = A\xi + BY + CZ + D$, the coefficients A, B, C and D can be calculated by applying the function on the four node points of a tetrahedron. This yields a system of equations for the coefficients of:

$$\begin{aligned}
A\xi_1 + BY_1 + CZ_1 + D &= \Phi_1 \\
A\xi_2 + BY_2 + CZ_2 + D &= \Phi_2 \\
A\xi_3 + BY_3 + CZ_3 + D &= \Phi_3 \\
A\xi_4 + BY_4 + CZ_4 + D &= \Phi_4
\end{aligned} \tag{A.1}$$

These equations are in the local coordinate system (ξ, Y, Z). By solving the above equations with Cramer's rule, the coefficients can be computes using the following:

$$\begin{aligned}
A &= L_1\Phi_1 + L_2\Phi_2 + L_3\Phi_3 + L_4\Phi_4 = \sum_{i=1}^{4} L_i\Phi_i \\
B &= M_1\Phi_1 + M_2\Phi_2 + M_3\Phi_3 + M_4\Phi_4 = \sum_{i=1}^{4} M_i\Phi_i \\
C &= N_1\Phi_1 + N_2\Phi_2 + N_3\Phi_3 + N_4\Phi_4 = \sum_{i=1}^{4} N_i\Phi_i \\
D &= O_1\Phi_1 + O_2\Phi_2 + O_3\Phi_3 + O_4\Phi_4 = \sum_{i=1}^{4} O_i\Phi_i
\end{aligned} \tag{A.2}$$

where,

$$\begin{aligned}
\Delta = {} & (Z_3 - Z_4)(\xi_1 Y_2 - \xi_2 Y_1) + (Z_4 - Z_2)(Z_2 - Z_3)(\xi_1 Y_4 - \xi_4 Y_1) \\
& + \xi_2 \left[Y_3 (Z_1 - Z_4) - Y_4 (Z_1 - Z_3)\right] \\
& + \xi_3 \left[Y_4 (Z_1 - Z_2) - Y_2 (Z_1 - Z_4)\right] \\
& + \xi_4 \left[Y_2 (Z_1 - Z_3) - Y_2 (Z_1 - Z_2)\right]
\end{aligned} \tag{A.3}$$

$$\begin{aligned}
L_1 &= \frac{Y_2(Z_3-Z_4)+Y_3(Z_4-Z_2)+Y_4(Z_2-Z_3)}{\Delta}\\
L_2 &= -\frac{Y_1(Z_3-Z_4)+Y_3(Z_4-Z_1)+Y_4(Z_1-Z_3)}{\Delta}\\
L_3 &= \frac{Y_1(Z_2-Z_4)+Y_2(Z_4-Z_1)+Y_4(Z_1-Z_2)}{\Delta}\\
L_4 &= -\frac{Y_1(Z_2-Z_3)+Y_2(Z_3-Z_1)+Y_3(Z_1-Z_2)}{\Delta}
\end{aligned} \tag{A.4}$$

$$\begin{aligned}
M_1 &= -\frac{\xi_2(Z_3-Z_4)+\xi_3(Z_4-Z_2)+\xi_4(Z_2-Z_3)}{\Delta}\\
M_2 &= \frac{\xi_1(Z_3-Z_4)+\xi_3(Z_4-Z_1)+\xi_4(Z_1-Z_3)}{\Delta}\\
M_3 &= -\frac{\xi_1(Z_2-Z_4)+\xi_2(Z_4-Z_1)+\xi_4(Z_1-Z_2)}{\Delta}\\
M_4 &= \frac{\xi_1(Z_2-Z_3)+\xi_2(Z_3-Z_1)+\xi_3(Z_1-Z_2)}{\Delta}
\end{aligned} \tag{A.5}$$

$$\begin{aligned}
N_1 &= \frac{\xi_2(Y_3-Y_4)+\xi_3(Y_4-Y_2)+\xi_4(Y_2-Y_3)}{\Delta}\\
N_2 &= -\frac{\xi_1(Y_3-Y_4)+\xi_3(Y_4-Y_1)+\xi_4(Y_1-Y_3)}{\Delta}\\
N_3 &= \frac{\xi_1(Y_2-Y_4)+\xi_2(Y_4-Y_1)+\xi_4(Y_1-Y_2)}{\Delta}\\
N_4 &= -\frac{\xi_1(Y_2-Y_3)+\xi_2(Y_3-Y_1)+\xi_3(Y_1-Y_2)}{\Delta}
\end{aligned} \tag{A.6}$$

$$\begin{aligned}
O_1 &= -\frac{\xi_2(Y_3Z_4-Y_4Z_3)+\xi_3(Y_4Z_2-Y_2Z_4)+\xi_4(Y_2Z_3-Y_3Z_2)}{\Delta}\\
O_2 &= \frac{\xi_1(Y_3Z_4-Y_4Z_3)+\xi_3(Y_4Z_1-Y_1Z_4)+\xi_4(Y_1Z_3-Y_3Z_1)}{\Delta}\\
O_3 &= -\frac{\xi_1(Y_2Z_4-Y_4Z_2)+\xi_2(Y_4Z_1-Y_1Z_4)+\xi_4(Y_1Z_2-Y_2Z_1)}{\Delta}\\
O_4 &= \frac{\xi_1(Y_2Z_3-Y_3Z_2)+\xi_2(Y_3Z_1-Y_1Z_3)+\xi_3(Y_1Z_2-Y_2Z_1)}{\Delta}
\end{aligned} \tag{A.7}$$

APPENDIX B. Shape Function for Pressure

Assuming the pressure varies linearly within each tetrahedral element, the following general equation can be used for the pressure:

$$p = -(\alpha x + \beta y + \gamma z + \eta) \tag{B.1}$$

Applying the general pressure equation to the node points yields:

$$\begin{aligned} \alpha x_1 + \beta y_1 + \gamma z_1 + \eta &= -p_1 \\ \alpha x_2 + \beta y_2 + \gamma z_2 + \eta &= -p_2 \\ \alpha x_3 + \beta y_3 + \gamma z_3 + \eta &= -p_3 \\ \alpha x_4 + \beta y_4 + \gamma z_4 + \eta &= -p_4 \end{aligned} \tag{B.2}$$

Using Cramer's rule, the Equations in B.2 are solved to yield the coefficients α, β, γ.

$$\begin{aligned} -\frac{\partial p}{\partial x} &= \alpha = \sum_{i=1}^{4} \bar{L}_i p_i \\ -\frac{\partial p}{\partial y} &= \beta = \sum_{i=1}^{4} \bar{M}_i p_i \\ -\frac{\partial p}{\partial z} &= \gamma = \sum_{i=1}^{4} \bar{N}_i p_i \end{aligned} \tag{B.3}$$

where, $\bar{L}_i, \bar{M}_i, \bar{N}_i$ are defined as follow:

$$\begin{aligned}\Delta = {} & x_1\left[y_2(z_3-z_4)+z_2(y_4-y_3)+(y_3z_4-y_4z_3)\right] \\ & - y_1\left[x_2(z_3-z_4)+z_2(x_4-x_3)+(x_3z_4-x_4z_3)\right] \\ & + z_1\left[x_2(y_3-y_4)+y_2(x_4-x_3)+(x_3y_4-x_4y_3)\right] \\ & - \left[x_2(y_3z_4-y_4z_3)+y_2(x_4z_3x_3z_4)+z_2(x_3y_4-x_4z_3)\right]\end{aligned} \tag{B.4}$$

$$\begin{aligned}\bar{L}_1 &= -\frac{y_2(z_3-z_4)+y_3(z_4-z_2)+y_4(z_2-z_3)}{\Delta} \\ \bar{L}_2 &= \frac{y_1(z_3-z_4)+y_3(z_4-z_1)+y_4(z_1-z_3)}{\Delta} \\ \bar{L}_3 &= -\frac{y_1(z_2-z_4)+y_2(z_4-z_1)+y_4(z_1-z_2)}{\Delta} \\ \bar{L}_4 &= \frac{y_1(z_2-z_3)+y_2(z_3-z_1)+y_3(z_1-z_2)}{\Delta}\end{aligned} \tag{B.5}$$

$$\begin{aligned}\bar{M}_1 &= \frac{x_2(z_3-z_4)+x_3(z_4-z_2)+x_4(z_2-z_3)}{\Delta} \\ \bar{M}_2 &= -\frac{x_1(z_3-z_4)+x_3(z_4-z_1)+x_4(z_1-z_3)}{\Delta} \\ \bar{M}_3 &= \frac{x_1(z_2-z_4)+x_2(z_4-z_1)+x_4(z_1-z_2)}{\Delta} \\ \bar{M}_4 &= -\frac{x_1(z_2-z_3)+x_2(z_3-z_1)+x_3(z_1-z_2)}{\Delta}\end{aligned} \tag{B.6}$$

$$\begin{aligned}\bar{N}_1 &= -\frac{x_2(y_3-y_4)+x_3(y_4-y_2)+x_4(y_2-y_3)}{\Delta} \\ \bar{N}_2 &= \frac{x_1(y_3-y_4)+x_3(y_4-y_1)+x_4(y_1-y_3)}{\Delta} \\ \bar{N}_3 &= -\frac{x_1(y_2-y_4)+x_2(y_4-y_1)+x_4(y_1-y_2)}{\Delta} \\ \bar{N}_4 &= \frac{x_1(y_2-y_3)+x_2(y_3-y_1)+x_3(y_1-y_2)}{\Delta}\end{aligned} \tag{B.7}$$

APPENDIX C. Pressure Equation Derivation

Starting with the continuity equation in terms of the artificial velocity field, the artificial velocity definitions are substituted into the equation and expanded.

$$\begin{aligned}
&\rho\Bigg[\frac{\hat{u}_r+\hat{u}_s+\hat{u}_t}{3}+\frac{d_r^u+d_s^u+d_t^u}{3}(S^u+\bar{L}_1p_1+\bar{L}_2p_2+\bar{L}_3p_3+\bar{L}_4p_4)n_x \\
&+\frac{\hat{v}_r+\hat{v}_s+\hat{v}_t}{3}+\frac{d_r^v+d_s^v+d_t^v}{3}(S^v+\bar{M}_1p_1+\bar{M}_2p_2+\bar{M}_3p_3+\bar{M}_4p_4)n_y \\
&+\frac{\hat{w}_r+\hat{w}_s+\hat{w}_t}{3}+\frac{d_r^w+d_s^w+d_t^w}{3}(S^w+\bar{N}_1p_1+\bar{N}_2p_2+\bar{N}_3p_3+\bar{N}_4p_4)n_z\Bigg]=0
\end{aligned} \tag{C.1}$$

To get the pressure coefficients, we assemble the terms around a particular point. Using node point 1 as the node of interest, all of the terms containing p_1 are collected and make up the a_P^p coefficient for the pressure equation. The rest of the terms containing pressures (p_2, p_3, p_4) contribute to the neighbor node coefficients a_{nb}^P. Finally, all remaining terms are collected to form the pressure source term b_p.

APPENDIX D. Runge-Kutta Profile

Execution time percentages of the primary subroutines for the CPU implementation of the Runge-Kutta Code.

Routine	Exec. time percentage
Momentum Coefficients	30.8%
Momentum Coefficient Assembly	27.6%
Pressure Coefficient Assembly	12.6%
Pressure Coefficients	11.8%
Median Dual Direction	7.6%
Pressure Solve	4.1%
Runge-Kutta Velocity Update	2.2%
Misc. Tasks	3.3%

Table D.1 Execution profile for the CPU implementation of the Runge-Kutta code.

BIBLIOGRAPHY

[1] Purohit, P., “Development of an explicit time accurate scheme for incompressible flows”, *MS Thesis Iowa State university*, Ames, IA, 2001

[2] Lestari, A., “Development of Unsteady Algorithms for Pressure-based Unstructured Solver for Two-Dimensional Incompressible Flows”, *MS Thesis, Iowa State University,*, Ames, IA, 2009

[3] Patankar, S. V., “Numerical Heat Transfer and Fluid Flow”, *Hemisphere Publishing Corp.*, New York, 1980

[4] Guntupalli, K., “Development, validation and verification of the Momentum Source Model for discrete rotor blades”, *MS Thesis Iowa State university*, Ames, IA, 2011

[5] Prakash, C., “An Improved Control Volume Finite-Element Method for Heat and Mass Transfer, and for Fluid Flow using Equal-Order Velocity-Pressure Interpolation”, *Numerical Heat Transfer*, 9:253-276, 1986

[6] Muir, B. L. and Baliga, B. R., “Solution of Three-Dimensional Convection-Diffusion Problems using Tetrahedral Elements and Flow-Oriented Upwind Interpolation Functions”, *Numerical Heat Transfer*, 9:143-162, 1986

[7] Prakash, C. and Patankar, S. V., “A Control Volume Based Finite Element Method for Solving the Navier-Stokes Equations using Equal-Order Velocity-Pressure Formulation”, *Numerical Heat Transfer*, 8:259-280, 1985

[8] NVIDIA Corporation., *NVIDIA CUDA C Programming Guide v4.2*, April 2012.

[9] Weiss, J. M. and Smith, W. A., "Solution of Unsteady, Low Mach Number Flow using a Preconditioned Multi-Stage Scheme on an Unstructured Mesh", *AIAA Computational Fluid Dynamics Conference, 11th, Orlando, FL*, July 6-9, 1993.

[10] Weiss, J. M. and Smith, W. A., "Preconditioning Applied to Variable and Density Time-Accurate Flows on Unstructured Meshes", *AIAA Journal*, 33:2050-2057, 1995

[11] Weiss, J. M., Maruszewski, J. P. and Smith, W. A., "Implicit Solution of Preconditioned Navier-Stokes Equations using Algebraic Multigrid", *AIAA Journal*, 37:29-36, 1999

[12] Chorin, A. J., "A Numerical Method for Solving Incompressible Viscous Flow Problems", *Journal of Computational Physics*, 2:12-26, 967

[13] Kwak, D., Chang, J. L. C. and Chakravarthy, S. R., "A Three-Dimensional Incompressible Navier-Stokes Flow Solver using Primitive Variables", *AIAA Journal*, 24:390-396, 1985

[14] Taylor, C. and Hood, P., "A Numerical Solution of the Navier-Stokes Equations using the Finite Element Technique", *Computational Fluids*, 1:73-100, 1973

[15] Gartling, D. K., "Finite Element Analysis of Viscous, Incompressible Fluid Flow", *PhD Thesis, University of Texas*, Austin, Texas, 1975

[16] Chung, T. J., "Finite Element Analysis in Fluid Dynamics", *McGraw-Hill*, New York, 1978

[17] Braaten, M. E., "Development and Evaluation of Iterative and Direct Methods for the Solution of Equations Governing Recirculating Flows", *PhD Thesis, University of Minnesota*, Minneapolis, MN, 1985

[18] Kim, J. and Moin, P. "Application of a Fractional-Step Method to Incompressible Navier-Stokes Equations", *Journal of Computational Physics*, 59:308-323, 1985

[19] Hughes, T. J. R., Liu, W. K. and Brooks, A., "Finite Element Analysis of Incompressible Viscous Flows by the Penalty Function Formulation", *Journal of Computational Physics*, 30:1-60, 1979

[20] Heinrich, J. C. and Marshall, R. S., "Viscous Incompressible Flow by a Penalty Function Finite Element Method", *Computational Fluids*, 9:73-83, 1981

[21] Reddy, J. N., "Penalty Finite Elements Methods in Mechanics", *AMD-vol. 51*, November 1982

[22] Braaten, M. E. and Shyy, W., "Comparison of Iterative and Direct Method for Viscous Flow Calculations in Body-Fitted Co-ordinates", *International Journal for Numerical Methods in Fluids*, 6:325-349, 1986

[23] Lee, S. L. and Tzong, R. Y., "Artificial Pressure for Pressure-Linked Equation", *International Journal for Heat Mass Transfer*, 35:2705-2716, 1992

[24] Sato, Y., Hino, T., and Hinatsu, M., "Unsteady Flow Simulation Around a Moving Body by an Unstructured Navier-Stokes Solver", *Proceedings of the Sixth Numerical Towing Tank Simulation*, Rome, Italy, 2003

[25] Sheng, C. H., Whitfield, D. L. and Anderson, W. K., "A Multiblock Approach for Calculating Incompressible Fluid Flows on Unstructured Grids", *AIAA Journal*, 37(2):169-176, 1999

[26] Harlow, F. H. and Welch, J. E., "Numerical Calculation of Three-Dimensional Time Dependent Viscous Incompressible Flow of Fluid with Free Surfaces", *Physics of Fluids*, 1:2182, 1965

[27] Moukalled, F. and Darwish, M., "A Unified Formulation of the Segregated Class of Algorithm for Fluid Flow at all Speeds", *Numerical Heat Transfer, Part B*, 37:103-139, 2000

[28] Miller, T. F. and Schmidt, F. W., "Use of Pressure Weighted Interpolation Method for Solution of Incompressible Navier-Stokes Equations on Non-staggered Grids", *Numerical Heat Transfer*, 14:213-233, 1988

[29] Majumdar,S., "Role of Under-relaxation in Momentum Interpolation for Calculation of Flow on Non-staggered Grids", *Numerical Heat Transfer*, 13:125-132, 1988

[30] Hall, C., Cavendish, J. and Frey, W., "The Dual-Variable Method for Solving Fluid Flow Difference Equations on Delaunay Triangulations", *Computational Fluids*, 20:145-164, 1991

[31] Hall, C., Porsching, T. and Hu, P., "Covolume-dual Variable Method for Thermally Expandable Flow on Unstructured Triangular Grids", *Journal of Computational Fluid Dynamics*, 2:111-139, 1994

[32] Nicolaides, R., Porsching, T. and Hall, C., "Covolume Methods in Computational Fluid Dynamics", *In M. Hafez, K. Oshima (Eds.), Computational Fluid Dynamics Review*, pages 279-299, Wiley, Chichester, UK, 1995

[33] Nicolaides, R., "The Covolume Approach on Computing Incompressible Flows", *In M. Gunzburger, R. Nicolaides (Eds.), Incompressible Computational Fluid Dynamics*, pages 295-333, Cambridge University Press, Cambridge, UK, 1993

[34] Perot, B., "Conservation Properties of Unstructured Staggered Mesh Schemes", *Journal of Computational Physics*, 159:58-89, 2000

[35] Rida, S., McKenty, F. Meng and Reggio, M., "A Staggered Control Volume Scheme for Unstructured Triangular Grids", *International Journal for Numerical Methods in Fluids*, 25:697-717, 1997

[36] Wenneker, I., Segal, A. and Wesseling, P., "A Mach-Uniform Unstructured Staggered Grid Method", *International Journal for Numerical Methods in Fluids*, 40:1209-1235, 2002

[37] Wenneker, I., Segal, A. and Wesseling, P., "Conservation Properties for a New Unstructured Staggered Scheme", *Computational Fluids*, 32:139-147, 2003

[38] Vidovic, D., Segal, A. and Wesseling, P., "A Superlinearly Convergent Finite Volume Method for the Incompressible Navier-Stokes Equations on Staggered Unstructured Grids", *Journal of Computational Physics*, 198:159-177, 2004

[39] Baliga, B. R. and Patankar, S. V., "A Control Volume Finite-Element Method for Two Dimensional Fluid Flow and Heat Transfer", *Numerical Heat Transfer*, 6:245-261, 1983

[40] Paraschivoiu, I., Mason, C., and Rajagopalan, R. G., "Predictions and Experiments of the VAWT Viscous Flow Field", *AIAA 87-1429, Presented at AIAA 19th Fluid Dynamics, Plasma Dynamics and Lasers Conference*, Honolulu, Hawaii, June 8-10, 1987

[41] Rajagopalan, R. G., and Zhang Zhaoxing, "Performance and Flow Field of a Ducted Propeller", *AIAA/ASME/SAE/ASEE 25th Joint Propulsion Conference*, Monterey, CA., July 10-12, 1989

[42] Rajagopalan, R. G., and Lim, C. K., "Laminar Flow Analysis of a Rotor in Hover", *Journal of the American Helicopter Society*, 36:12-23, 1991

[43] Rajagopalan, R. G., and Mathur, S. R., "Three Dimensional Analysis of a Rotor in Forward Flight", *Journal of the American Helicopter Society*, Vol. 38, No. 3, pp. 14-25, 1993

[44] Zori, L., and Rajagopalan, R. G., "Navier-Stokes Calculations of Rotor-Airframe Interaction in Forward Flight", *Journal of the American Helicopter Society*, Vol. 40, No. 2, pp. 56-67, 1995

[45] Rajagopalan, R. G., Berg, D. E., and Klimas, P. C., "Development of a Three-Dimensional Model for the Darrieus Rotor and Its Wake", *Journal of Propulsion and Power*, Vol. 11, No. 2, pp. 185-195, March-April 1995

[46] Kim, Y. H., and Park, S. O., "Navier-Stokes Simulation of Unsteady Rotor-Airframe Interaction with Momentum Source Method", *International Journal of Aeronautical and Space Sciences*, Vol. 10, No. 2, November 2009

[47] Yang, J. Y., Yang, S. C., Chen, Y. N. and Hsu, C. A., "Implicit Weighted ENO Schemes for the Three-Dimensional Incompressible Navier-Stokes Equations", *Journal of Computational Physics*, Vol. 146, pp. 464-487, 1998