ABSTRACT

LOU, JIALIN. Development and Implementation of Reconstructed Discontinuous Galerkin Methods for Computational Fluid Dynamics on GPUs. (Under the direction of Hong Luo.)

Recently, the application of general-purpose graphics processing unit (GPGPU) technology to the computational fluid dynamics (CFD) solvers has been popular. With GPGPU, one can benefit from an exciting opportunity to significantly accelerate the CFD solvers by offloading the compute-intensive portions of the application to the GPU, while the remainder of the program still runs on the CPU.

The discontinuous Galerkin (DG) finite element methods, originally introduced for the solution of neutron transport equation in 1973, has become popular for CFD research for the last few decades. To overcome the high computing costs associated with the DG methods, the reconstructed discontinuous Galerkin (rDG) methods using a Taylor basis have been developed for the solution of the compressible Euler/Navier-Stokes equations on arbitrary grids.

The objective of the effort presented in this Ph.D. work is to port an unstructured CFD solver, reconstructed discontinuous Galerkin flow solver (RDGFLO), onto GPU platform using OpenACC. The solver is based on a third-order hierarchical Weighted Essentially Non-Oscillatory (WENO) reconstructed DG methods. By taking advantages of the OpenACC parallel programming model, the presented scheme requires the minimum code intrusion and algorithm alteration to upgrade a legacy CFD solver without much extra time and effort in programming, resulting in a unified portable code for both CPU and GPU platforms.

A hyperbolic rDG method based on first order hyperbolic system (FOHS) is developed in this work. In addition to the least-squares reconstruction, the variational reconstruction scheme is also adopted to obtain arbitrary higher-order moments with compact data structure, leading to a more robust method. By combining the advantages of the FOHS formulation and the rDG methods, an effort has been made to develop a more reliable, accurate, efficient, and robust method for solving some model equations including diffusion equations and advection-diffusion equations.

In this study, both underlying DG(P\textsubscript{1}) scheme and rDG(P\textsubscript{1}P\textsubscript{2}) scheme which indicates that a quadratic polynomial solution is obtained from the underlying linear polynomial DG solution via a hierarchical WENO reconstruction, are ported onto GPGPU platform. Both multi-stage explicit Runge-Kutta and simple implicit backward Euler methods are implemented for time advancement. Additionally, for the unsteady case solved by the newly developed hyperbolic rDG scheme, an explicit first stage, single diagonal coefficient, diagonally implicit Runge-Kutta scheme (ESDIRK) has also been employed. Meanwhile, p-multigrid technique is also adopted in the study to accelerate the convergence. The analytical differentiation and automatic differen-
tiation (AD) are utilized to obtain the flux Jacobian matrices for implicit time marching. The linearized system would be solved using the lower-upper symmetric Gauss-Seidel (LU-SGS) pre-conditioned general minimum residual (GMRES) algorithm on the CPU and symmetric Gauss-Seidel (SGS) relaxation method on the GPU. Considering the fact that GPU is a shared memory device, several approaches have been made to deal with the “race condition” and also the efficiency. A face coloring algorithm is adopted to eliminate the memory contention because of the threading of internal and boundary face integrals. The resulting linearized system would be solved using lower-upper symmetric Gauss-Seidel (LU-SGS) or symmetric Gauss-Seidel (SGS) on the GPGPU platform. A similar element reordering algorithm needs to be employed to resolve the inherent data dependency. Also, a fine-grained style algorithm for matrix inversion needs to be adopted for efficient GPU computation. With the help of a message passing interface (MPI) programming paradigm, multi-GPU computing ability is obtained, where the METIS library is used for the partitioning of a mesh into subdomain meshes of approximately the same size.

A number of inviscid and viscous flow problems are presented to verify the implementation of the developed schemes on the GPU. Strong scaling tests are carried out to compare the unit running time on single GPU and single CPU to obtain the speedup factor of the developed methods. Also, weak scaling tests are used for several cases to test the parallel efficiency for multi-GPU computing by comparing the unit running time with different number of GPU cards for an approximately fixed problem size per GPU card. The results of timing measurements indicate that this OpenACC-based parallel scheme is able to significantly accelerate the solving procedure for the equivalent legacy CPU code. Numerical experiments of the model equations for the newly developed hyperbolic rDG methods demonstrate that the presented methods are able to achieve the designed optimal order of accuracy for both solutions and their derivatives on regular, irregular, and heterogeneous grids. From the comparison, the presented schemes outperform the conventional diffusive DG methods like BR2 and DDG in terms of the magnitude of the error, the order of accuracy, the size of time steps, and the CPU times required to achieve steady state solutions, indicating that the developed hyperbolic rDG methods provide an attractive and probably an even superior alternative for discretizing the diffusive fluxes.

In summary, the developed GPU accelerated rDG method and the hyperbolic rDG schemes have a great potential to become a viable, attractive, competitive and ultimately superior high-order methods in the field of computational fluid dynamics.
Development and Implementation of Reconstructed Discontinuous Galerkin Methods for Computational Fluid Dynamics on GPUs

by

Jialin Lou

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Aerospace Engineering

Raleigh, North Carolina

2018

APPROVED BY:

Jack R. Edwards Jr.          Hassan A. Hassan

Zhilin Li
Minor Member

Hong Luo
Chair of Advisory Committee
DEDICATION

To my family, whose love and support over the years make this work possible. Especially to my father who could not see this thesis completed due to his death from cancer at the age of 53. We will always miss him and try our best to live up to his expectation of us. May he rest in peace.
BIOGRAPHY

The author was born in Zhangzhou, a city in Fujian Province, China. He received the Bachelor’s degree of engineering in Engineering Mechanics at Beijing Institute of Technology (BIT), Beijing, China, in June 2013. Later, the author was enrolled in the direct path from BS to Ph.D. program of Aerospace Engineering with minor in Mathematics-GM, in the Department of Mechanical and Aerospace Engineering at North Carolina State University (NCSU), Raleigh, North Carolina, USA since August 2013. His academic advisor is Dr. Hong Luo. He received an en-route master of science degree in Aerospace Engineering in May 2016.
ACKNOWLEDGEMENTS

First of all, I would like to express my thanks to my academic advisor Dr. Hong Luo for his great support and help with my research work within the past few years at North Carolina State University.

I would also like to give my appreciation to my committee members Dr. Jack R. Edwards, Dr. Hassan A. Hassan from the Department of Mechanical and Aerospace Engineering, and Dr. Zhilin Li from the Department of Mathematics for their time of review on my dissertation and valuable suggestions.

Special thanks should go to Dr. Frank Muller from the Department of Computer Science for his help and suggestions in the aspect of computer science. Most of the simulation of my research work is completed in the ARC cluster managed by his team.

I sincerely thank Dr. Hiroaki Nishikawa from National Institute of Aerospace, for all the support he provided during our discussion and his help for the paper writing.

Meanwhile, the thanks should also go to my former group members, Dr. Yidong Xia and Dr. Lixiang Luo. They provided a lot of help in my research, especially for the GPU parallel computing technics. I would also like to take this opportunity to thank my former and current group member, including Dr. Xiaodong Liu, Dr. Chuanjin Wang, Dr. Lijun Xuan, Dr. Xiaoquan Yang, Aditya Kiran Pandare, Lingquan Li, Chad Rollins, and Aditya Kashi, for their suggestion, assistance, and encouragement with my research work. Their time and effort is sincerely appreciated.

Last but not least, I would like to thank my family and my friends, for their love, support, and blessings throughout my doctoral student life.
# TABLE OF CONTENTS

## LIST OF TABLES

<table>
<thead>
<tr>
<th>Chapter 1 Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2 Governing Equations of Fluid Dynamics</td>
<td>15</td>
</tr>
<tr>
<td>Chapter 3 Reconstructed Discontinuous Galerkin Spatial Discretization</td>
<td>21</td>
</tr>
<tr>
<td>Chapter 4 Hyperbolic Reconstructed Discontinuous Galerkin Methods</td>
<td>53</td>
</tr>
<tr>
<td>Chapter 5 Temporal Integration Methods</td>
<td>70</td>
</tr>
</tbody>
</table>

## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Chapter 1 Introduction</th>
<th>xi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2 Governing Equations of Fluid Dynamics</td>
<td>x</td>
</tr>
</tbody>
</table>

## Chapter 1 Introduction

1.1 General-Purpose Graphics Processing Unit for Computational Fluid Dynamics 1
1.2 Reconstructed Discontinuous Galerkin Methods 3
1.3 Motivation and Goals 12

## Chapter 2 Governing Equations of Fluid Dynamics

2.1 Navier-Stokes Equations 15
2.2 Euler Equations 17
2.3 Nondimensionalization 18

## Chapter 3 Reconstructed Discontinuous Galerkin Spatial Discretization

3.1 Discontinuous Galerkin Methods 21
3.1.1 Weak Formulation 21
3.1.2 Basis Functions 23
3.2 Reconstructed Discontinuous Galerkin Methods 28
3.2.1 Least-Squares Reconstruction 29
3.2.2 Variational Reconstruction 34
3.2.3 WENO Reconstruction at P<sub>2</sub>: WENO(P<sub>1</sub>P<sub>2</sub>) 35
3.2.4 WENO Reconstruction at P<sub>1</sub>: HWENO(P<sub>1</sub>P<sub>2</sub>) 37
3.3 Numerical Flux 40
3.3.1 Inviscid Flux Scheme 40
3.3.2 Viscous Flux Scheme 42
3.4 Numerical Integration 44
3.5 Boundary Conditions 48
3.5.1 Characteristic Boundary 48
3.5.2 Slip Wall / Symmetry Boundary 50
3.5.3 No-Slip Adiabatic Wall Boundary 51
3.5.4 No-Slip Isothermal Wall Boundary 51
3.5.5 Periodic Boundary 52

## Chapter 4 Hyperbolic Reconstructed Discontinuous Galerkin Methods

4.1 First Order Hyperbolic System Formulation 53
4.2 Weak Formulation 58
4.3 Spatial Discretization 60
4.4 Numerical Flux 66
4.5 Boundary Condition 68

## Chapter 5 Temporal Integration Methods

5.1 Explicit Scheme 71
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>Three-stage Third-order TVD Runge-Kutta Scheme</td>
<td>71</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Four-stage Fourth-order Runge-Kutta Scheme</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>Implicit Scheme</td>
<td>72</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Backward Euler Formulation</td>
<td>72</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Explicit First Stage, Single Diagonal Coefficient, Diagonally Implicit Runge-Kutta Scheme</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>$p$-multigrid Method</td>
<td>75</td>
</tr>
<tr>
<td>5.4</td>
<td>Solution of the Linear System of Equations</td>
<td>76</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Jacobian Matrix</td>
<td>77</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Linear Solver</td>
<td>89</td>
</tr>
</tbody>
</table>

Chapter 6 OpenACC Based Parallelization ....................................................... 91
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Overview</td>
<td>91</td>
</tr>
<tr>
<td>6.2</td>
<td>Face Coloring Method</td>
<td>93</td>
</tr>
<tr>
<td>6.3</td>
<td>Gauss-Jordan Elimination based Matrix Inversion</td>
<td>97</td>
</tr>
<tr>
<td>6.4</td>
<td>Parallelization for Linear Solver</td>
<td>100</td>
</tr>
<tr>
<td>6.5</td>
<td>Multiple-GPU Computation</td>
<td>101</td>
</tr>
</tbody>
</table>

Chapter 7 Numerical Results ........................................................................ 106
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Flow Simulation</td>
<td>107</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Subsonic Flow past a Sphere</td>
<td>108</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Transonic Flow over a Boeing 747 Aircraft</td>
<td>119</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Laminar Flow past a Sphere</td>
<td>126</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Quasi-2D Lid Driven Square Cavity</td>
<td>130</td>
</tr>
<tr>
<td>7.1.5</td>
<td>3D Taylor-Green Vortex</td>
<td>137</td>
</tr>
<tr>
<td>7.2</td>
<td>Model Equations</td>
<td>143</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Steady Scalar Diffusion Case without Source Term</td>
<td>144</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Steady Scalar Diffusion Case with Source Term</td>
<td>152</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Steady Tensor Diffusion Case with Source Term</td>
<td>154</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Steady Nonlinear Diffusion Case</td>
<td>157</td>
</tr>
<tr>
<td>7.2.5</td>
<td>1D Boundary Layer Case</td>
<td>160</td>
</tr>
<tr>
<td>7.2.6</td>
<td>2D Steady Advection-Diffusion Case</td>
<td>164</td>
</tr>
<tr>
<td>7.2.7</td>
<td>1D Unsteady Advection-Diffusion Case</td>
<td>173</td>
</tr>
<tr>
<td>7.2.8</td>
<td>2D Unsteady Advection-Diffusion Case</td>
<td>175</td>
</tr>
</tbody>
</table>

Chapter 8 Conclusions ................................................................................... 177
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Summary of Completed Work</td>
<td>177</td>
</tr>
<tr>
<td>8.2</td>
<td>Outlook of Future Work</td>
<td>179</td>
</tr>
</tbody>
</table>

REFERENCES ................................................................................................. 180

APPENDICES ............................................................................................... 193
<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A</td>
<td>Relationship between the Variational Reconstruction and Compact Finite Difference Scheme</td>
<td>194</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Various Types of Elements</td>
<td>196</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>B.1</td>
<td>Triangular Elements (two dimensions)</td>
<td>196</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Systems of Coordinates</td>
<td>196</td>
</tr>
<tr>
<td>B.1.2</td>
<td>3-Node Linear Triangle</td>
<td>198</td>
</tr>
<tr>
<td>B.1.3</td>
<td>6-Node Curvilinear Triangle</td>
<td>199</td>
</tr>
<tr>
<td>B.2</td>
<td>Quadrilateral Elements (two dimensions)</td>
<td>201</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Systems of Coordinates</td>
<td>201</td>
</tr>
<tr>
<td>B.2.2</td>
<td>4-Node Bilinear Quadrilateral</td>
<td>201</td>
</tr>
<tr>
<td>B.2.3</td>
<td>8-Node Curvilinear Quadrilateral</td>
<td>201</td>
</tr>
<tr>
<td>B.3</td>
<td>Tetrahedral Elements</td>
<td>203</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Systems of Coordinates</td>
<td>203</td>
</tr>
<tr>
<td>B.3.2</td>
<td>4-node Linear Tetrahedron</td>
<td>204</td>
</tr>
<tr>
<td>B.3.3</td>
<td>10-node Curvilinear Tetrahedron</td>
<td>205</td>
</tr>
<tr>
<td>B.4</td>
<td>Hexahedral Elements</td>
<td>207</td>
</tr>
<tr>
<td>B.4.1</td>
<td>8-Node Trilinear Hexahedron</td>
<td>207</td>
</tr>
<tr>
<td>B.4.2</td>
<td>20-Node Curvilinear Hexahedron</td>
<td>208</td>
</tr>
<tr>
<td>B.5</td>
<td>Prismatic Elements</td>
<td>211</td>
</tr>
<tr>
<td>B.5.1</td>
<td>6-Node Prism</td>
<td>211</td>
</tr>
<tr>
<td>B.5.2</td>
<td>15-node Prism</td>
<td>212</td>
</tr>
<tr>
<td>B.6</td>
<td>Pyramidal Elements</td>
<td>214</td>
</tr>
<tr>
<td>B.6.1</td>
<td>5-Node Pyramid</td>
<td>214</td>
</tr>
<tr>
<td>B.6.2</td>
<td>13-Node Pyramid</td>
<td>215</td>
</tr>
</tbody>
</table>

Appendix C  Tables of Gaussian Quadrature Formulas  217
C.1 Triangle  218
C.2 Quadrilateral  219
C.3 Tetrahedron  220
C.4 Hexahedron  221
C.5 Prism  222
C.6 Pyramid  223
LIST OF TABLES

Table 2.1  Reference variables for nondimensionalization of the governing equations. \hspace{1cm} 18

Table 3.1  Cost analysis for different numerical methods on a tetrahedral grid. \hspace{1cm} 37
Table 3.2  Cost analysis for different numerical methods on a hexahedral grid. \hspace{1cm} 38

Table 4.1  Comparison of DG schemes and hyperbolic-DG/rDG schemes for expected order of accuracy. \hspace{1cm} 66

Table 5.1  A simple source program. \hspace{1cm} 85
Table 5.2  A simple differential source program. \hspace{1cm} 86
Table 5.3  Example: implementation of TAPENADE on RDGFLO. \hspace{1cm} 87
Table 5.4  Pseudo code: contribution of face integrals for the block diagonal matrix $D$, lower matrix $L$, and upper matrix $U$. \hspace{1cm} 88
Table 5.5  Flowchart of the GMRES algorithm. \hspace{1cm} 90

Table 6.1  An example of loop over the elements. \hspace{1cm} 92
Table 6.2  An example of loop over the edges. \hspace{1cm} 96
Table 6.3  Gauss-Jordan elimination without pivoting for an $n \times n$ matrix. \hspace{1cm} 98
Table 6.4  ACC version of GJE without pivoting for $Nelem$ matrices of size $n \times n$. \hspace{1cm} 99
Table 6.5  Workflow for the main loop over the explicit time iterations. \hspace{1cm} 102
Table 6.6  Workflow for the data exchange and synchronization procedure. \hspace{1cm} 103

Table 7.1  Part of the Nvidia GPGPU resource on the NCSU’s ARC cluster (donated by NVIDIA). \hspace{1cm} 107
Table 7.2  Discretization errors and convergence rates obtained on the three successively refined tetrahedral grids for inviscid subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$. \hspace{1cm} 110
Table 7.3  Timing measurements of using TVD$RK3+rDG(P_1P_2)$ methods for subsonic flow past a sphere. \hspace{1cm} 114
Table 7.4  Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for using rDG methods with TV$DK3$ for subsonic flow past a sphere. \hspace{1cm} 115
Table 7.5  Timing measurements of using $p$-multigrid DG/rDG methods for subsonic flow past a sphere. \hspace{1cm} 115
Table 7.6  Timing measurements of using BDF1+DG/rDG methods for subsonic flow past a sphere. \hspace{1cm} 117
Table 7.7  Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for using rDG methods with BDF1 for subsonic flow past a sphere. \hspace{1cm} 118
Table 7.8  Timing measurements of using TVD$RK3+rDG(P_1P_2)$ methods for inviscid transonic flow over a Boeing 747 aircraft $M_\infty = 0.85, \alpha = 2^\circ$. \hspace{1cm} 121
Table 7.9  Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for using rDG methods with TVDRK3 for transonic flow over a Boeing 747 aircraft. ........................................ 121
Table 7.10  Timing measurements of using p-multigrid DG methods for transonic flow past a Boeing 747 aircraft. ................................................................. 122
Table 7.11  Timing measurements of using BDF1+DG/rDG methods for transonic flow past a Boeing 747 aircraft. ................................................................. 123
Table 7.12  Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for using rDG methods with BDF1 for transonic flow over a Boeing 747 aircraft. ........................................ 124
Table 7.13  Timing measurements of using TVDRK3+rDG(P1P2) methods for laminar flow past a sphere. ................................................................. 128
Table 7.14  Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for using rDG methods with TVDRK3 for laminar flow past a sphere. ........................................ 128
Table 7.15  Timing measurements of using TVDRK3+rDG(P1P2) methods for quasi-2D lid driven square cavity problem at Re = 10,000. .......................... 132
Table 7.16  Timing measurements of weak scaling obtained on a cluster of Nvidia Tesla C2050 GPU cards for a quasi-2D lid driven square cavity. .............. 135
Table 7.17  Timing measurements of using RK4+rDG(P1P2) methods for Taylor-Green vortex problem ................................................................. 140
Table 7.18  Timing measurements of weak scaling obtained on a cluster of Nvidia Tesla C2050 GPU cards for a Taylor-Green vortex problem ......................... 140
Table 7.19  Order of accuracy on different type of grids for the steady nonlinear diffusion case. ................................................................. 159
Table 7.20  Order of accuracy for the 1D boundary layer case with different Re. .......................... 161
Table 7.21  Order of accuracy for the 2D steady advection-diffusion case on regular grids with different \( \nu \). ................................................................. 165
Table 7.22  Order of accuracy for the 2D steady advection-diffusion case on irregular grids with different \( \nu \). ................................................................. 166
Table 7.23  Order of accuracy for 2D steady advection-diffusion case on heterogeneous grids with different \( \nu \). ................................................................. 166
Table 7.24  Timing measurements of using hyperbolic DG/rDG methods for the quasi-2D steady advection-diffusion case on regular grids. ....................... 171
Table 7.25  Timing measurements of using hyperbolic DG/rDG methods for the quasi-2D steady advection-diffusion case on irregular grids. ....................... 171
Table 7.26  Timing measurements of using hyperbolic DG/rDG methods for the quasi-2D steady advection-diffusion case on heterogeneous grids. ....................... 172
Table 7.27  Order of accuracy for the 2D unsteady advection-diffusion case. .......................... 176

Table B.1  Abbreviation and description for various types of elements. ................................................................. 197
Table B.2  Shape functions for the 3-node triangle and their derivatives. ................................................................. 199
Table B.3  Shape functions for the 6-node triangle and their derivatives. ................................................................. 200
Table B.4  Shape functions for the 4-node quadrilateral and their derivatives. ................................................................. 202
Table B.5  Shape functions for the 8-node quadrilateral and their derivatives. ................................................................. 202
Table B.6  Shape functions for the 4-node tetrahedron and their derivatives.  . . . . . 204
Table B.7  Shape functions for the 10-node tetrahedron and their derivatives.  . . . . . 206
Table B.8  Shape functions for the 8-node hexahedron and their derivatives.  . . . . . 207
Table B.9  Shape functions for the 6-node prism and their derivatives.  . . . . . . . 211
Table B.10 Shape functions for the 15-node prism and their derivatives.  . . . . . . . . 213
Table B.11 Shape functions for the 5-node pyramid and their derivatives.  . . . . . . . . 214
Table B.12 Shape functions for the 13-node pyramid.  . . . . . . . . . . . . . . . . . . . . 216

Table C.1  Integration constants of Gaussian quadrature rules for a triangle.  . . . . . 218
Table C.2  Integration constants of Gaussian quadrature rules for a quadrilateral.  . . 219
Table C.3  Integration constants of Gaussian quadrature rules for a tetrahedron.  . . . 220
Table C.4  Integration constants of Gaussian quadrature rules for a hexahedron.  . . 221
Table C.5  Integration constants of Gaussian quadrature rules for a prism.  . . . . . . 222
Table C.6  Integration constants of Gaussian quadrature rules for a pyramid.  . . . . . 223
LIST OF FIGURES

Figure 3.1 Representation of polynomial solutions using finite element shape functions: (a) $Q_1/P_1$; (b) $Q_2/P_2$. ................................. 23
Figure 3.2 Representation of polynomial solutions using a Taylor series expansion. ........................................ 25
Figure 3.3 Representation of the four face-neighboring cells surrounding a tetrahedron (a) and the six face-neighboring cells surrounding a hexahedron (b). 32
Figure 3.4 Transformation of an element in $(x,y)$ physical space into a canonical element reference $(\xi,\eta)$ space. ................................. 44
Figure 6.1 Domain decomposition by METIS for a tetrahedral grid (only surface shown) for flow past a Boeing 747 aircraft, 16 partitions. .......... 104
Figure 7.1 The NVIDIA Tesla GPGPUs used in the present work: (a) C2050 (448 stream processors, 3GB memory); (b) K20c (2496 stream processors, 5GB memory). ......................................................... 107
Figure 7.2 A sequence of three successively globally refined tetrahedral grids for inviscid subsonic flow past a sphere. .............................. 110
Figure 7.3 Computed density contours in the flow field by the DG($P_1$) (a–c) and rDG($P_1P_2$) (d–f) on the tetrahedral grids for inviscid subsonic flow past a sphere at $M_\infty = 0.5$. ................................. 111
Figure 7.4 Computed pressure contours in the flow field by the DG($P_1$) (a–c) and rDG($P_1P_2$) (d–f) on the tetrahedral grids for inviscid subsonic flow past a sphere at $M_\infty = 0.5$. ................................. 112
Figure 7.5 Computed Mach number contours in the flow field by the DG($P_1$) (a–c) and rDG($P_1$) (d–f) on the tetrahedral grids for inviscid subsonic flow past a sphere at $M_\infty = 0.5$. ................................. 113
Figure 7.6 Plot of the timing measurements for inviscid subsonic flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using TVDRK3+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards. ................................. 116
Figure 7.7 Subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$: (a) Comparison of convergence history versus CPU time between TV-DRK3 and $p$-multigrid methods for DG($P_1$); (b) Comparison of convergence history versus CPU time between TVDRK and $p$-multigrid methods for rDG($P_1P_2$). ................................. 116
Figure 7.8 Averaged one main time loop run times (in seconds) for running implicit rDG($P_1P_2$) on the finest mesh for the inviscid sphere case. ................................. 117
Figure 7.9 Plot of the timing measurements for inviscid subsonic flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using BDF1+DG/rDG schemes: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards. ................................. 119
Figure 7.10 Inviscid transonic flow over a Boeing 747 aircraft computed using rDG(P_1P_2), (a) the tetrahedral grid used for computation (Nelem = 253, 577, Npoin = 48, 851, Nbfac = 23, 616), (b) computed density contours, (c) computed Mach number contours, and (d) computed pressure contours.

Figure 7.11 Plot of the timing measurements for inviscid transonic flow over a Boeing 747 aircraft with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using TVDRK3+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.

Figure 7.12 Transonic flow past a Boeing 747 aircraft at a free-stream Mach number of $M_\infty = 0.85$ and a angle of attack of $\alpha = 2^\circ$: (a) Comparison of convergence history versus CPU time between TVDRK and $p$-multigrid methods for DG(P_1); (b) Comparison of convergence history versus CPU time between TVDRK and $p$-multigrid methods for rDG(P_1P_2).

Figure 7.13 Plot of the timing measurements for transonic flow over a Boeing 747 aircraft with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using BDF1+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.

Figure 7.14 (a) Plot of the surface meshes of the tetrahedral grid for laminar flow past a sphere at $M_\infty = 0.5$ and $Re = 118$; (b) plot of the computed velocity streamtraces on the symmetry plane using rDG(P_1P_2).

Figure 7.15 Plot of the computed contours obtained by rDG(P_1P_2) on the surface meshes in the flow field at $M_\infty = 0.5$ and $Re = 118$: (a) pressure contours; (b) Mach number contours.

Figure 7.16 Plot of the timing measurements for laminar flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using TVDRK3+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.

Figure 7.17 A quasi-2D lid driven square cavity flow at a lid velocity of $v_B = (0.2, 0, 0)$ and a series of Reynolds number: (a) The sparse hexahedral grid ($32 \times 32 \times 2$ grid points) used in the verification test, (b) $Re = 100$, (c) $Re = 1000$, and (d) $Re = 10,000$.

Figure 7.18 Profiles of the normalized velocity components $u/u_B$ and $v/u_B$ on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid driven square cavity at a lid velocity of $v_B = (0.2, 0, 0)$, and a Reynolds number of $Re = 100$.

Figure 7.19 Profiles of the normalized velocity components $u/u_B$ and $v/u_B$ on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid driven square cavity at a lid velocity of $v_B = (0.2, 0, 0)$, and a Reynolds number of $Re = 1000$. 
Figure 7.20 Profiles of the normalized velocity components \( u/u_B \) and \( v/u_B \) on a sparse hexahedral grid (32 × 32 × 2 grid points) for a quasi-2D lid driven square cavity at a lid velocity of \( v_B = (0.2, 0, 0) \), and a Reynolds number of \( Re = 10,000 \). .......................... 134

Figure 7.21 Plot of the timing measurements for a quasi-2D lid driven square cavity with a fixed problem size (approximately half million elements) per GPU card (Nvidia Tesla C2050): (a) Unit running time versus the number of GPU cards, (b) Parallel efficiency versus the number of GPU cards ........... 136

Figure 7.22 The isosurfaces of Q2 criterion colored by velocity magnitude at time \( t = 8t_c \). .......................................................... 139

Figure 7.23 Evolution of the dimensionless kinetic energy as a function of the dimensionless time using RK4. .................................................. 140

Figure 7.24 Evolution of the dimensionless kinetic energy dissipation rate as a function of the dimensionless time using RK4. ......................... 141

Figure 7.25 Evolution of the dimensionless enstrophy rate as a function of the dimensionless time using RK4. ................................. 141

Figure 7.26 Plot of the timing measurements for a 3D Taylor-Green Vortex with a fixed problem size (approximately half million elements) per GPU card (Nvidia Tesla C2050): (a) Unit running time versus the number of GPU cards. (b) Parallel efficiency versus the number of GPU cards. ............. 142

Figure 7.27 The second mesh of every type, that is, 17 × 17 regular grid, 17 × 17 irregular grid, and 23 × 21 heterogeneous grid. ....................... 143

Figure 7.28 Distribution of \( R \) in the coarsest irregular mesh (left) and heterogeneous mesh (right). ..................................................... 144

Figure 7.29 Grid refinement study of the hyperbolic rDG methods for the steady scalar diffusion case without source term on regular grids using implicit BDF1 ............................................. 146

Figure 7.30 Grid refinement study of the hyperbolic rDG methods for the steady scalar diffusion case without source term on regular grids using implicit BDF1 ............................................. 147

Figure 7.31 Grid refinement study of the hyperbolic rDG methods for the steady scalar diffusion case without source term on heterogeneous grids using implicit BDF1 ............................................. 147

Figure 7.32 Comparison between hyperbolic rDG methods and BR2 scheme in terms of order of accuracy for the steady scalar diffusion case without source term on regular grids using explicit TVDRK3. ......................... 148

Figure 7.33 Comparison between hyperbolic rDG methods and BR2 scheme in terms of order of accuracy for the steady scalar diffusion case without source term on irregular grids using explicit TVDRK3. ......................... 148

Figure 7.34 Comparison between hyperbolic rDG methods and BR2 scheme in terms of order of accuracy for the steady scalar diffusion case without source term on heterogeneous grids using explicit TVDRK3. ......................... 149
Figure 7.35 Comparison between hyperbolic rDG methods and BR2 scheme in terms of CPU time for the steady scalar diffusion case without source term on 17 × 17 regular grids using explicit TVDRK3. ........................................... 149
Figure 7.36 Comparison between hyperbolic rDG methods and BR2 scheme in terms of CPU time for the steady scalar diffusion case without source term on 17 × 17 irregular grids using explicit TVDRK3. ................................. 150
Figure 7.37 Comparison between hyperbolic rDG methods and BR2 scheme in terms of CPU time for the steady scalar diffusion case without source term on 23 × 21 heterogeneous grids using explicit TVDRK3. ..................... 150
Figure 7.38 Time steps and CPU time required to reach the steady state for regular grids using explicit TVDRK3. .................................................. 151
Figure 7.39 Grid refinement study of the steady scalar diffusion case with source term on regular grids using implicit BDF1. ................................. 152
Figure 7.40 Grid refinement study of the steady scalar diffusion case with source term on irregular grids using implicit BDF1. ................................. 153
Figure 7.41 Grid refinement study of the steady scalar diffusion case with source term on heterogeneous grids using implicit BDF1. .................... 153
Figure 7.42 Grid refinement study of the steady tensor diffusion case with source term on regular grids using implicit BDF1. ................................. 155
Figure 7.43 Grid refinement study of the steady tensor diffusion case with source term on irregular grids using implicit BDF1. ................................. 155
Figure 7.44 Grid refinement study of the steady tensor diffusion case with source term on heterogeneous grids using implicit BDF1. ................. 156
Figure 7.45 Grid refinement study of the steady nonlinear diffusion case on regular grids using implicit BDF1. .................................................. 157
Figure 7.46 Grid refinement study of the steady nonlinear diffusion case on irregular grids using implicit BDF1. .................................................. 158
Figure 7.47 Grid refinement study of the steady nonlinear diffusion case on heterogeneous grids using implicit BDF1. ................................. 159
Figure 7.48 Grid refinement study for the 1D boundary layer case, Re= 10^{-8}. 162
Figure 7.49 Grid refinement study for the 1D boundary layer case, Re= 1. 162
Figure 7.50 Grid refinement study for the 1D boundary layer case, Re= 10^8. 163
Figure 7.51 Grid refinement study for the 2D steady advection-diffusion case on regular grids with ν = 10^{-8}, Re = \sqrt{5} \times 10^8. 167
Figure 7.52 Grid refinement study for the 2D steady advection-diffusion case on regular grids with ν = 1, Re = \sqrt{5}. 167
Figure 7.53 Grid refinement study for the 2D steady advection-diffusion case on regular grids with ν = 10^8, Re = \sqrt{5} \times 10^{-8}. 168
Figure 7.54 Grid refinement study for the 2D steady advection-diffusion case on irregular grids with ν = 10^{-8}, Re = \sqrt{5} \times 10^8. 168
Figure 7.55 Grid refinement study for the 2D steady advection-diffusion case on irregular grids with ν = 1, Re = \sqrt{5}. 169
Figure 7.56 Grid refinement study for the 2D steady advection-diffusion case on irregular grids with ν = 10^8, Re = \sqrt{5} \times 10^{-8}. 169
Figure 7.57 Grid refinement study for the 2D steady advection-diffusion case on heterogeneous grids with $\nu = 10^{-8}, \text{Re} = \sqrt{5} \times 10^8$. ................................. 170
Figure 7.58 Grid refinement study for the 2D steady advection-diffusion case on heterogeneous grids with $\nu = 1, \text{Re} = \sqrt{5}$. ................................................. 170
Figure 7.59 Grid refinement study for the 2D steady advection-diffusion case on heterogeneous grids with $\nu = 10^8, \text{Re} = \sqrt{5} \times 10^{-8}$. ................................. 171
Figure 7.60 Grid refinement study on regular grids for the 1D unsteady advection-diffusion case. ................................................................. 174
Figure 7.61 Comparison between the numerical solutions obtained by the hyperbolic rDG methods and DDG methods for the 1D unsteady advection-diffusion case, $t_{\text{end}} = 10^{-3}$ (5 periods). ................................................. 174
Figure 7.62 Grid refinement study on regular grids for 2D unsteady advection-diffusion case. ................................................................. 175
Figure 7.63 Grid refinement study on irregular grids for 2D unsteady advection-diffusion case. ................................................................. 176

Figure B.1 Representation of the 3-node triangular element. ................. 197
Figure B.2 Representation of barycentric coordinates for a triangle in physical space. 198
Figure B.3 Representation of barycentric coordinates for a reference triangle. .. 198
Figure B.4 Representation of the 6-node triangular element. .................. 199
Figure B.5 Representation of the 4-node quadrilateral element. ............... 201
Figure B.6 Representation of the 8-node quadrilateral element. ............... 202
Figure B.7 Representation of the 4-node tetrahedral element. ................. 203
Figure B.8 Representation of barycentric coordinates for a tetrahedron in physical space. ................................................................. 204
Figure B.9 Representation of the 10-node tetrahedral element. ............... 205
Figure B.10 Representation of the 8-node hexahedral element. ............... 207
Figure B.11 Representation of the 20-node hexahedral element. ............... 208
Figure B.12 Representation of the 6-node prismatic element. ................. 211
Figure B.13 Representation of the 15-node prismatic element. ............... 212
Figure B.14 Representation of the 5-node pyramidal element. ............... 214
Figure B.15 Representation of the 13-node pyramidal element. ............... 215
Chapter 1

Introduction

1.1 General-Purpose Graphics Processing Unit for Computational Fluid Dynamics

Nowadays, with increasing attention in science and engineering field, the General-Purpose Graphics Processing Unit (GPGPU [123]) technology, which is essentially a shared memory vector device, offers a new opportunity to dramatically accelerate the CPU-based code by offloading compute-intensive portions of the application to the GPU, while the remainder of the computer program still runs on the CPU, which also make it expected to be a major compute unit in the near future. From a user’s perspective, the solvers simply run much faster.

Computational Fluid Dynamics (CFD), a branch of mechanics that applies numerical way to solve fluid dynamics problem, has been one of the most significant applications on supercomputers. The presence of GPGPU could outperform the traditional CPU based parallel computing and therefore to meet the needs to solving complex simulation CFD problem [7, 21, 22, 30, 32, 33, 50, 60, 61, 106, 127, 140].

Among the vendors of GPGPU hardware and software, NVIDIA has been an exceptional pioneer in promoting and leading the development of GPGPU technology for the past decade. As a popular parallel programming model and platform in GPGPU technology, NVIDIA’s CUDA application programming interface (API) and CUDA-enabled accelerators has drawn many researchers’ attention. To the best of the authors’ knowledge, numerical methods in CFD solvers that have been attentively studied based on CUDA include the finite difference (FD) methods, spectral difference (SD) methods, finite volume (FV) methods, discontinuous Galerkin methods, Lattice Boltzmann methods and more. For example, Elsen et al. [44] reported a 3D high-order
FD method solver for large calculation on multiblock structured grids, Klöckner et al. [67] developed a 3D unstructured high-order nodal DG method solver for the Maxwell’s equations, Corrigan et al. [34] proposed a 3D FV solver for compressible inviscid flow on unstructured tetrahedral grids and Zimmerman et al. [162] presented a SD method solver for the Navier-Stokes equations on unstructured hexahedral grids. Nevertheless, the development of CUDA capabilities extended from an existing CFD solver is not a trivial job, because people have to define an explicit layout of the threads on the GPU (numbers of blocks and numbers of threads) for each kernel function [63]. Such a project often requires tremendous hours in programming, as developers have to rewrite all the core content of the source code. Moreover, for a production-level solver, people also need to address both the short-term and long-term investment concerns like the cost and profit, as well as platform portability. These factors can often set people back from investing on GPU computing for their well-established solution products. Even a research-oriented CFD solver is concerned, people may be more inclined to maintain compatibility of their codes across multiple platforms, instead of pursuing performance on one particular platform at the price of being unable to run their codes on other mainstream platforms. Therefore, the development strategy of a CFD solver based on one unique model like CUDA might be a risky long-term investment with unclear prospect of the vendor’s own plan. Fortunately, NVIDIA CUDA is not the sole player in this area. Two other programming models on GPU include OpenCL [139], the currently dominant open GPGPU programming model (but dropped from further discussion because it does not support the FORTRAN programming language) and OpenACC [149], a new programming standard for parallel computing developed by Cray, CAPS, NVIDIA and PGI.

The OpenACC standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems as well as to closely resemble the OpenMP standard: people simply need to annotate their code to identify the areas that should be accelerated by wrapping with the OpenACC directives and some runtime library routines, without huge effort to change the original algorithms as to accommodate the code to a specific GPU architecture and compiler. With the OpenACC directives, people benefit not only from easy implementation but also from the freedom to compile the very same code and conduct computations on either CPU or GPU from different vendors. However, compared with CUDA in terms of many desired technical features, the OpenACC standard still lags behind because of vendors’ distribution plan (note that NVIDIA is among the OpenACC’s main supporters). Nevertheless, OpenACC is quickly maturing as an attractive, future GPU parallel programming model for developing portable computer codes and offers a promising approach to minimize the investment in legacy CFD solver by presenting an easy migration path to accelerated computing. Support of OpenACC
is available in the commercial compilers from PGI, Cray, and CAPS. OpenUH is an Open64-based open source OpenACC compiler, developed by HPCTools group from the University of Houston. In addition, the GCC (GNU Compiler Collection) project team is also working toward supporting OpenACC in the GCC compilers.

A typical GPU has hundreds or even thousands of computation cores. However, compared with a typical CPU core, those on GPU card would have much less computation power and local cache memory. Therefore, for optimal performance on GPU, the algorithm should be divided into smaller units, thus occupying more cores with the same amount of work. In addition, the amount of data each core processes should be kept as small as possible. The latter aspect is particularly important for solving a block-sparse system, which consists of a large amount of square sub-matrices of an identical size. On each sub-matrix, if the operation is mapped to one GPU core, the algorithm easily becomes heavily memory-bound, especially when the dimension of the sub-matrix is large, resulting in serious performance penalty. In other words, the key to achieve higher speed up factor on GPU platform is fine granularity.

Nevertheless, it is generally difficult to port the implicit algorithm to GPU platform. First of all, the size of sub-matrices would cause the memory-bound issue, especially for higher order methods. Secondly, it is not straightforward to utilize some popular linear solver or preconditioner like Symmetric Gauss-Seidel (SGS) or Lower Upper-Symmetric Gauss-Seidel (LU-SGS) due to the inherent data dependency. Additionally, the iterative solver like Generalized Minimal Residual (GMRES) method would require additional storage for some auxiliary arrays, which would bring challenge to GPU computing since the local cache memory of GPGPU is limited.

### 1.2 Reconstructed Discontinuous Galerkin Methods

In mathematics, the discontinuous Galerkin (DG) methods form a class of numerical methods for solving differential equations, and have recently become popular for the solution of systems of conservation laws. In particular, they have received considerable interest for problems with a dominant first-order part, e. g. in fluid mechanics, electrodynamics, and plasma physics. The methods were first proposed and analyzed in the early 1970s as a technique to numerically solve partial differential equations. In 1973, Reed and Hill [133] introduced a DG method to solve the hyperbolic neutron transport equation. In 1974, Le Saint and Raviart [71] analyzed the DG methods for linear hyperbolic problems, first derived the priori error estimates and proved rates of convergence. Nowadays, they are widely used in computational fluid dynamics (CFD), computational acoustics, and computational electromagnetics. The discontinuous Galerkin methods combine two advantageous features commonly associated to the finite element (FE) methods
and finite volume (FV) methods. As in classical finite element methods, accuracy is obtained by
means of high-order polynomial approximation within an element rather than by wide stencils
as in the case of finite volume methods. The physics of wave propagation is, however, accounted
for by solving the Riemann problems [141] that arise from the discontinuous representation of
the solution at element interfaces. In this respect, the methods are therefore similar to finite
volume methods. A more comprehensive overview of the discontinuous Galerkin methods is
given by Cockburn et al. [27].

The discontinuous Galerkin methods are attractive to many researchers due to the fact
that they have many promising features. First of all, the DG methods have several useful
mathematical properties with respect to conservation, stability and convergence. Also, they
can be easily extended to higher-order (> 2nd) approximation. Additionally, the DG methods
are well suited for complex geometries since they can be applied on unstructured grids. In
addition, the methods can also handle non-conforming elements, where the grids are allowed to
have hanging nodes. As for parallelization, since DG methods are compact and each element
is independent, they are highly parallelizable [69,92]. Since the elements are discontinuous,
and the inter-element communications are minimal, domain decomposition can be effectively
employed. The compactness also allows for structured and simplified coding for the methods.
Meanwhile, since refining or coarsening a grid can be achieved without considering the continuity
restriction commonly associated with the conforming elements, they allow easy implementation
of hp-refinement [18,55,108,121,148], for example, the order of accuracy, or shape, can vary
from element to element. Last but not least, the DG methods have the ability to compute low
Mach number flow problems [10,86] without recourse to the time-preconditioning techniques
normally required for the finite volume methods.

In spite of the enormous advances in the theoretical and numerical analysis of the DG
methods [4, 5, 6, 11, 12, 13, 15, 26, 28, 29, 31, 55, 56, 56, 65, 66, 68, 95, 121, 124, 126, 129, 130, 131,
143,144], the DG methods have a number of weaknesses that have yet to be addressed, before
they can be robustly used for flow problems of practical interest in complex configuration
environment. In particular, how to effectively control spurious oscillations in the presence of
strong discontinuities, how to reduce the computing costs, and how to efficiently solve elliptic
problems or discretize diffusion terms in the parabolic equations remain three most challenging
and unresolved issues for the DG methods.

Similar to any other high-order schemes (> 1st order), the discontinuous Galerkin methods
will suffer from non-physical oscillations in the vicinity of discontinuities that exist in problems
governed by hyperbolic conservations laws. Usually, either a discontinuity capturing technique
or an appropriate slope limiter would be applied to addressing this issue. The former approach
adds explicitly consistent artificial viscosity terms to the discontinuous Galerkin discretization. The main disadvantage of this method is that it usually requires some user-specified parameters, which can be both grid and problem dependent. As for the slope limiting, the classical techniques are not directly applicable for high-order DG methods because of the presence of volume integral terms in the formulation. Therefore, the slope limiter is not integrated in the computation of the residual, but effectively acts as a post-processing filter. Many slope limiters used in the finite volume methods can then be used or modified to meet the needs of the DG methods. Unfortunately, the use of the limiters will reduce the order of accuracy to first order in the presence of discontinuities. Furthermore, the active limiters in the smooth extrema will pollute the solution in the flow field and ultimately destroy the higher-order accuracy of the DG methods. Indeed, it is not an exaggeration to state that the design of efficient, effective, and robust limiters is one of the bottlenecks in the development of the DG methods for solving the conservation laws. Most efforts in the development of the DG methods have primarily been focused on the exploration of their advantages such as higher-order spatial discretization, posteriori error estimation, adaptive algorithms, and parallelization.

Indeed, compared with FV methods, the DG methods require the solution of systems of equations with more unknowns (number of degree of freedom) for the same grids. Consequently, the DG methods have been recognized as expensive in terms of both computational cost and storage requirement. Note that the temporal discretization methods have lagged far behind compared with the DG spatial discretization. Usually, explicit temporal discretizations such as multi-stage TVD (total variation diminishing) Runge-Kutta schemes [11, 12, 26, 28, 29] are used to advance the solution in time. In general, explicit schemes and their boundary conditions are easy to implement, vectorize and parallelize, and require only limited memory storage. However, for large-scale simulations and especially for high-order solutions, the rate of convergence slows down dramatically, resulting in inefficient solution techniques to steady-state solutions. To speed up the convergence, a multigrid strategy [57, 87, 90, 91, 146] or an implicit method is required. In general, implicit methods require the solution of a linear system of equations arising from the linearization of a fully implicit scheme at each timestep or iteration. For the past fifteen years, various authors have contributed remarkable efforts to the development of efficient implicit solution strategies for the DG methods [13, 14, 15, 35, 45, 66, 124, 132, 159]. All these implicit DG methods achieved a significant speedup of convergence in contrast to their explicit counterpart. Unfortunately, the drawback is that they require a considerable amount of memory to store the Jacobian matrix, which may be prohibitive for large-scale problems and high-order solutions. Even in the implementation of the so-called matrix-free implicit methods [84, 96], where only a block diagonal matrix is required to store, the memory
requirements can still be extremely demanding. The block diagonal matrix requires a storage of \((N_{\text{degr}} \times N_{\text{Netot}}) \times N_{\text{Netot}} \times N_{\text{elem}}\), where \(N_{\text{degr}}\) is the number of degree of freedom for the polynomial (3 for \(P_1\), 6 for \(P_2\), and 10 for \(P_3\) for triangular element in 2D; 4 for \(P_1\), 10 for \(P_2\), and 20 for \(P_3\) for tetrahedral element in 3D), \(N_{\text{Netot}}\) is the number of components in the solution vector (4 for 2D, and 5 for the three-dimensional Navier-Stokes equations), and \(N_{\text{elem}}\) is the number of elements for the grid. For example, for a 4th-order (cubic polynomial finite element approximation \(P_3\)) DG method in 3D, the storage of this block diagonal matrix alone requires 10,000 words per element.

In order to reduce the high computing costs associated with the DG methods, Dumbser et al. [39, 41, 42] have introduced a new family of reconstructed DG methods, termed \(P_n P_m\) schemes and referred to as rDG\((P_n P_m)\) in this work, where \(P_m\) indicates that a piecewise polynomial of degree of \(n\) is used to compute the fluxes and source term. The rDG\((P_n P_m)\) schemes are designed to enhance the accuracy of the discontinuous Galerkin method by increasing the order of the underlying polynomial solution. The beauty of the rDG\((P_n P_m)\) schemes is that they provide a unified formulation for both the finite volume and DG methods, and contain both the classical finite volume and standard DG methods as two special cases of the rDG\((P_n P_m)\) schemes, and thus allow for a direct efficiency comparison. When \(n = 0\), i.e., a piecewise constant polynomial is used to represent a numerical solution, the rDG\((P_0 P_m)\) scheme is nothing but the classical high-order finite volume scheme, where a polynomial solution of degree \(m\) \((m \geq 1)\) is reconstructed from a piecewise constant solution. When \(m = n\), the reconstruction reduces to the identity operator, and the rDG\((P_n P_n)\) scheme yields a standard DG method.

Traditionally, DG methods would either use the standard Lagrange or hierarchical node-based finite element basis functions to represent numerical polynomial solutions in each element. As a result, the unknowns to be solved are the variables at the nodes and the polynomial solutions are dependent on the shape of elements. In the present work, the numerical polynomial solutions are represented using a Taylor series expansion at the centroid of the cell, which can be further expressed as a combination of cell-averaged variables and their derivatives at the centroid of the cell. The unknowns to be solved in this formulation are cell-averaged values and their derivatives at the center of the cells, regardless of the element shapes. In other words, by using this formulation, the DG method can be easily implemented on arbitrary grids. Therefore, the numerical method based on this formulation has the ability to compute the 1D, 2D, and 3D problems using the very same code, which greatly alleviates the need and pain for code maintenance and upgrade. Also, due to the fact that the basis functions are hierarchic, the implementation of \(p\)-multigrid and \(p\)-refinement methods has become straightforward.

Obviously, the construction of an accurate and efficient reconstruction operator is crucial to
the success of the rDG(PnPm) schemes. In Dumbser’s work, a higher-order polynomial solution is reconstructed using an $L^2$ projection, requiring it indistinguishable from the underlying DG solutions in the contributing cells in the weak sense. The resulting over-determined system is then solved by using a least-squares method that guarantees exact conservation, not only of the cell averages but also of all higher-order moments in the reconstructed cell itself, such as slopes and curvatures. However, this conservative least-squares reconstruction approach is computationally expensive, as the $L^2$ projection, i.e., the operation of integration, is required to obtain the resulting over-determined system. Furthermore, the reconstruction might be problematic for a boundary cell, where the number of the adjacent face-neighboring cells might not be enough to provide the necessary information to recover a polynomial solution of a desired order. Fortunately, the projection-based reconstruction is not the only way to obtain a polynomial solution of higher order from the underlying discontinuous Galerkin solutions. In the reconstructed DG method using a Taylor basis [92, 93, 94, 97, 99] developed by Luo et al. for the solution of the compressible Euler/Navier-Stokes equations on arbitrary grids, a higher-order polynomial solution is reconstructed by using a strong interpolation, requiring point values and derivatives to be interpolated on the adjacent face-neighboring cells. The resulting over-determined linear system of equations is then solved in the least-squares sense. This reconstruction scheme only involves the von Neumann neighborhood, and thus is compact, simple, robust, and flexible. Like the projection-based reconstruction, the strong reconstruction scheme guarantees exact conservation, not only of the cell averages but also of their slopes due to a judicious choice of the Taylor basis. More recently, Zhang et al. [160, 161] presented a class of hybrid DG/FV methods for the conservation laws, where the second derivatives in a cell are obtained from the first derivatives in the cell itself and its neighboring cells using a Green-Gauss reconstruction widely used in the finite volume methods. This also provides a fast, simple, and robust way to obtain higher-order polynomial solutions. More recently, Luo et al. [98, 100] have conducted a comparative study for these three reconstructed discontinuous Galerkin methods rDG(P1P2) for solving the 2D Euler equations on arbitrary grids. It is found that all the three reconstructed discontinuous Galerkin methods can deliver the desired third-order accuracy and significantly improve the accuracy of the underlying second-order DG method, although the least-squares reconstruction method provides the best performance in terms of both accuracy and robustness.

However, the attempt to directly extend the least-squares rDG method to solve the 3D Euler equations on tetrahedral grids is not successful. Like the second-order cell-centered finite volume methods, i.e., rDG(P0P1), the resultant rDG(P1P2) method is numerically unstable. Although the rDG(P0P1) methods are in general stable in 2D and on Cartesian or structured grids in 3D, they suffer from the so-called linear instability on unstructured tetrahedral grids,
when the reconstruction stencils only involve von Neumann neighborhood, i.e., adjacent face-neighboring cells \[51\]. Unfortunately, the least-squares \(r\text{DG}(P_1P_2)\) method exhibits the same linear instability, which can be overcome by using extended reconstruction stencils. However, this is achieved at the expense of sacrificing the compactness of the underlying DG methods. Furthermore, these linear reconstruction-based DG methods will suffer from the non-physical oscillations in the vicinity of strong iscontinuities for the compressible Euler equations. Alternatively, the essentially non-oscillatory (ENO), weighted essentially non-oscillatory (WENO) and Hermite-WENO schemes can be used to reconstruct a higher-order polynomial solution, which can not only enhance the order of accuracy of the underlying DG method but also achieve both the linear and non-linear stability.

The ENO schemes were initially introduced by Harten \textit{et al.} \[54\], in which oscillations up to the order of the truncation error are allowed to overcome the drawbacks and limitations of limiter-based schemes. ENO schemes avoid interpolation across high-gradient regions through biasing of the reconstruction. This biasing is achieved by reconstructing the solution on several stencils at each location and selecting the reconstruction which is in some sense the smoothest. This allows ENO schemes to retain higher-order accuracy near high-gradient regions. However, the selection process can lead to convergence problems and loss of accuracy in regions with smooth solution variations. To counter these problems, the so-called weighted ENO (WENO) scheme introduced by Liu \textit{et al.} \[75\] is designed to present better convergence rate for steady-state problems, better smoothing for the flux vectors, and better accuracy using the same stencils than the ENO scheme. The WENO scheme uses a suitably weighted combination of all reconstructions rather than just the one that is judged to be the smoothest. The weighting is designed to favor the smooth reconstruction in the sense that its weight is small if the oscillation of a reconstructed polynomial is high, and its weight is order of one if a reconstructed polynomial has low oscillation. Qiu and Shu initiated the use of the WENO scheme as limiters for the DG method \[131\] for solving the 1D and 2D Euler equations on structured grids. Later on, they constructed a class of WENO schemes based on Hermite polynomials, termed as Hermite-WENO schemes and applied these schemes as limiters for the DG methods \[129, 130\]. The main difference between Hermite-WENO and WENO schemes is that the former has a more compact stencil than the latter for the same order of accuracy.

Unfortunately, implementation of both ENO and WENO schemes is fairly complicated on arbitrary grids, especially in 3D. In fact, there are very few results obtained using ENO/WENO on unstructured grids in 3D especially for higher-order reconstruction. Harten and Chakravarthy \[53\], Abgrall \[2\], and Sonar \[138\] presented the first implementation of ENO schemes on unstructured triangular grids. Implementation of WENO schemes on unstructured triangular grids
was also presented by Friedrich [46] and Hu et al. [59].

The Hermite-WENO schemes has been developed on 1D and 2D structured grids for the DG methods by Balsara et al [8], where the Hermite-WENO reconstruction scheme is relatively simple and straightforward. In the work presented, a Taylor basis [97]reconstruction-based DG method, $rDG(P_1P_2)$, based on a Hierarchical WENO reconstruction scheme, termed as HWENO$(P_1P_2)$ [99], is developed for the solution of the compressible Euler and Navier-Stokes equations on single-type and hybrid unstructured grids in 3D [150]. This reconstructed DG method is designed not only to reduce the high computing costs of the DG methods, but also to avoid spurious oscillations in the vicinity of strong discontinuities, thus effectively overcoming two of the three most severe shortcomings of the DG methods and ensuring the linear and non-linear stability of the reconstructed DG method. In this $rDG(P_1P_2)$ method, a quadratic solution is obtained through the HWENO$(P_1P_2)$ reconstruction in three steps: (1) all the second derivatives ($P_2$) in each cell are first reconstructed using the solution variables ($P_0$) and their first derivatives ($P_1$) from adjacent face-neighboring cells via a least-squares method; (2) a WENO reconstruction would be performed to obtain the final second derivatives in each cell based on the reconstructed second derivatives in the cell itself and its adjacent face-neighboring cells; (3) the gradients of the quadratic polynomial solutions are then modified using a WENO reconstruction. As one can see, the linear stability of the $rDG$ method is ensured through step 2 while the non-physical oscillations in the vicinity of strong discontinuities is eliminated in step 3 and thus to ensure the non-linear stability of the developed scheme. This reconstruction scheme, by taking advantage of handily available and yet valuable information, namely the gradients in the context of the DG methods, only involves von Neumann neighborhood and thus is compact, simple, robust and flexible.

Another reconstruction scheme has been developed recently based on variational formulation for finite volume method [147] and can be regarded as an extension of the compact finite difference schemes [70] to the unstructured grids. Different from the least squares approach, this variational reconstruction (VR) would solve an extreme value problem to obtain the higher moments of the solution. The problem minimizes the jumps of the values of the reconstructed polynomial solutions and their spatial derivatives at cell interfaces, and therefore maximizes smoothness of the reconstructed polynomial solutions. This reconstruction schemes has many attracting properties, including the $k$-exactness and the robustness. Unlike the LS method, the VR method is stable even on the tetrahedral grids, since its stencils are intrinsically the entire mesh. Unlike $rDG(P_nP_m)$ method based on the least square method, $rDG_{VR}(P_nP_m)$ can easily extend to higher order of $m$ due to the fact that it only require the compact data structure but has global stencil. Moreover, this would also make it easier for parallelization. In this work,
rDG(P_nP_m) stands for the reconstruction DG method based on least squares method.

As the underlying DG(P_1) method is second order, and the basis functions are at most linear functions, fewer integration points are then required for both volume and face integrals, and the number of unknowns (the number of degrees of freedom) remains the same as for the DG(P_1) method. Consequently, this rDG(P_1P_2) method is more efficient than its third-order DG(P_2) counterpart.

The DG methods are indeed a natural choice for the solution of the hyperbolic equations, such as the compressible Euler equations. However, the DG formulation is far less certain and advantageous for elliptic problems or parabolic equations such as the compressible Navier-Stokes equations, where diffusive fluxes exist and which require the evaluation of the solution derivatives at the interfaces. Taking a simple arithmetic mean of the solution derivatives from the left and right is inconsistent, because it does not take into account a possible jump of the solutions. A number of numerical methods have been proposed in the literature to address this issue, such as those by Bassi and Rebay [11, 14, 15], Cockburn and Shu [29], Baumann and Oden [18], Peraire and Persson [124], and many others. Arnold et al. [6] have analyzed a large class of DGM for second-order elliptic problems in a unified formulation. All these methods have introduced in some way the influence of the discontinuities in order to define correct and consistent diffusive fluxes. Gassner et al. [47] introduced a numerical scheme based on the exact solution of the diffusive generalized Riemann problem for the DGM. Liu et al. [73, 74] proposed a direct discontinuous Galerkin (DDG) method to solve diffusion problems based on the direct weak formulation for solutions of parabolic equations. Cheng et al. [25] successfully extended the DDG method to solve the compressible Navier-Stokes equations on arbitrary grids. Luo et al. have developed a reconstructed discontinuous Galerkin method using an inter-cell reconstruction [94] for the solution of the compressible Navier-Stokes equations. Unfortunately, all these methods seem to require substantially more computational effort than the classical continuous finite element methods, which are naturally more suited for the discretization of elliptic problems. There is also an approach where a scalar diffusion scheme is derived from a hyperbolic diffusion formulation [112, 113]. It has been extended to higher-order in the context of the residual-distribution method [3], but has not been extended in the DG methods beyond second-order.

As a matter of fact, over the last several years, an alternative approach to viscous discretizations, which reformulates the viscous terms as a first-order hyperbolic system (FOHS), was developed by Nishikawa [77, 103, 107, 109, 110, 114, 115, 117, 120]. In the FOHS formulation, by including derivative quantities as additional variables, the equations are first formulated as a first order system (FOS). Then, it is rendered to be hyperbolic, which is the distinguished
feature of the FOHS method from other FOS methods, by adding pseudo time derivatives to the first-order system. It thus generates a system of pseudo-time evolution equations for the solution and the derivatives in the partial differential equation (PDE) level, not in the discretization level as in DG methods. The hyperbolic reformulation in the PDE level would allow a dramatic simplification in the discretization as the well-established methods can be directly applied to the viscous terms. The FOHS method is especially attractive in the context of the DG methods since it allows the use of inviscid algorithms for the viscous terms and thus greatly simplifies the discretization of the compressible Navier-Stokes equations. Moreover, the FOHS method yields a numerical scheme that can achieve the same order of accuracy in the solution and its derivatives on irregular grids and high-quality noise-free gradients on such grids. This is a very important feature for unstructured-grid viscous simulations, in which target quantities are derivatives, e.g., viscous stresses and heat fluxes.

A challenge in combining the DG method and the FOHS method lies in a very large number of discrete unknowns arising from both methods. For a scalar equation in two dimensions, the FOHS method introduces two derivatives as additional variables, and a P₁ DG method introduces three degrees of freedom (DoFs) for each variable (solution, and two derivatives), resulting in the total of nine degrees of freedom. In 2015, Nishikawa noticed that these degrees of freedom can be significantly reduced by unifying inter-related high-order moments of the derivative variables and extending the idea of Scheme-II [114] to replace high-order moments of a solution polynomial by the derivative variables. He has shown that the total number of degrees of freedom can be reduced from nine to six while the order of polynomial is upgraded to quadratic for the solution variable. The resulting approximation is comparable to a conventional P₂ DG method. Therefore, if compared with a one-order higher conventional DG method, the FOHS method requires virtually no increase in the degrees of freedom. The method extends systematically to arbitrary order of accuracy: Pₖ hyperbolic DG method gives comparable accuracy as Pₖ⁺₁ DG method for the same number of degrees of freedom. Later, the method was presented formally in Ref. [104], focusing on advection-dominated problems. However, the specific method described in Ref. [104] is not yet an attractive approach for practical applications. First, it has one-order-lower accuracy in the diffusion term than a conventional DG method (see Table 3 in Ref. [104]), thus leading to lower order accuracy, for example, in boundary layer calculations. Second, since a direct solver is employed for solving the linear system in the Newton method, convergence acceleration by the elimination of second derivatives, which is one of the advantages of the hyperbolic method, is not achieved. Therefore, this approach is, although more efficient than a straightforward DG discretization of the FOHS, actually less efficient than conventional DG methods, not fully taking advantage of the hyperbolic method.
More importantly, the method does not contribute to reducing the cost of the DG method. In this study, we explore the combination of the FOHS method and the rDG method in order to reduce the cost of the DG method towards affordable high-order unstructured-grid methods for practical applications.

Another difficulty would arise when it comes to unsteady problems. Typically, implicit-time stepping schemes are employed in the hyperbolic method, and all previous developments rely on the backward difference formulas (BDF) [79,120]. The first- and second-order BDF formulas are unconditionally stable (L-stable), and thus suitable for practical applications. However, higher-order ($\geq 3$) backward-difference formulas are only conditionally stable. It is highly desirable to develop unconditionally-stable high-order hyperbolic schemes for unsteady problems. Also, the high-order BDF method is not self-starting, requiring several lower order BDF methods at the starting stages. Furthermore, the time step would need to be fixed unless some further modification is made, like the variable time step BDF methods [120]. To overcome these difficulties, we consider an explicit first stage, single diagonal coefficient, diagonally implicit Runge-Kutta time integration scheme (ESDIRK) [19] and demonstrate the unsteady capability of the developed hyperbolic schemes. Compared with BDF methods, implicit Runge-Kutta (IRK) methods are A-stable and L-stable for arbitrary order in time. Also, variable time step sizes can be easily applied. Moreover, ESDIRK schemes are self-starting, i.e., one does not need to set up different temporal schemes at the beginning. Although ESIRK schemes would be more computationally expensive than the BDF counterpart for the same time step size, the cost can be reduced by taking a larger time step without encountering instability and maintaining the design order of accuracy.

1.3 Motivation and Goals

The objective of the effort presented in this Ph.D. work is to develop and port a 3D legacy code, reconstructed discontinuous Galerkin Flow solver (RDGFLO) for the compressible Navier-Stokes equations, onto GPU platforms using OpenACC. This third-order accurate rDG method is based on a hierarchical weighted essentially non-oscillatory reconstruction scheme, termed as HWENO($P_1P_2$) to indicate that a quadratic polynomial solution is obtained from the underlying linear polynomial DG solution via a hierarchical WENO reconstruction. The HWENO($P_1P_2$) is designed not only to enhance the accuracy of the underlying DG($P_1$) method but also to ensure non-linear stability of the rDG method. In this reconstruction scheme, a quadratic polynomial ($P_2$) solution is first reconstructed using a least-squares approach from the underlying linear ($P_1$) discontinuous Galerkin solution. The final quadratic solution is then obtained using
a Hermite WENO reconstruction, which is necessary to ensure the linear stability of the rDG method on 3D unstructured grids. The first derivatives of the quadratic polynomial solution are then reconstructed using a WENO reconstruction in order to eliminate spurious oscillations in the vicinity of strong discontinuities, thus ensuring the non-linear stability of the rDG method. By taking advantages of the OpenACC parallel programing model, the presented scheme requires the minimum code intrusion and algorithm alteration to upgrade a legacy CFD solver without much extra time of effort in programming, resulting a unified portable code for both CPU and GPU platforms.

In this work, hyperbolic rDG methods based on first order hyperbolic system (FOHS) are developed. Both least squares reconstruction and variational reconstruction has been employed to obtain higher order numerical solutions while remaining the total degrees of freedom relatively small. By combining the advantages of the FOHS formulation and the rDG methods, an effort has been made to develop a more reliable, accurate, efficient, and robust method for solving some model equations including diffusion equation, advection-diffusion equation and ultimately the incompressible and compressible Navier-Stokes equations on fully irregular, adaptive, anisotropic, unstructured grids.

In this study, both underlying DG(P$_1$) scheme and rDG(P$_1$P$_2$) scheme which indicates that a quadratic polynomial solution is obtained from the underlying linear polynomial DG solution via a hierarchical WENO reconstruction, are ported onto GPGPU platform. Both multi-stage explicit Runge-Kutta and simple implicit backward Euler methods are implemented for time advancement. Additionally, for the unsteady case solved by the newly developed hyperbolic rDG scheme, implicit Runge-Kutta has also been employed. Meanwhile $p$-multigrid technics are also adopted in the study to accelerate the convergence. For $p$-multigrid scheme, DG(P$_1$) and rDG(P$_1$P$_2$) are implemented as higher level method while DG(P$_0$) scheme are used as lower level method to provide correction to speed up the convergence. Note that the lower level scheme are marched using implicit backward Euler methods. A face coloring algorithm is adopted to eliminate the memory contention because of the threading of internal and boundary face integrals. In the implicit method, the analytical differentiation and automatic differentiation (AD) are developed and implemented to obtain the resulting flux Jacobian matrices. Also, a fine-grained style algorithm for matrix inversion need to be adopted for efficient GPU computation. The resulting linearized system would be solved using lower-upper symmetric Gauss-Seidel (LU-SGS) or symmetric Gauss-Seidel (SGS) on the GPGPU platform. A similar element reordering algorithm needs to be employed to resolved the inherent data dependency. With the help of a message passing interface (MPI) programming paradigm, multi-GPU computing ability is obtained, where the METIS library is used for the partitioning of a mesh into subdomain
meshes of approximately the same size.

A number of inviscid and viscous flow problems are presented to verify the implementation of the developed scheme. Strong scaling tests are carried out to compare the unit running time on single GPU and single CPU to obtain the speed up factor of the developed methods. Also, weak scaling tests are carried out for several cases to test the parallel efficiency for multi-GPU computing by comparing the unit running time with different number of GPU cards for an approximately fixed problem size per GPU card. The results of timing measurements indicate that this OpenACC-based parallel scheme is able to significantly accelerate the solving procedure for the equivalent legacy CPU code. Numerical experiments of the model equations for the developed hyperbolic rDG methods demonstrate that the presented methods are able to achieve the designed optimal order of accuracy for both solutions and their derivatives on regular, irregular, and heterogeneous grids, and outperform the conventional diffusive DG method like BR2, DDG, in terms of the magnitude of the error, the order of accuracy, the size of time steps, and the CPU times required to achieve steady state solutions, indicating that the developed hyperbolic rDG methods provide an attractive and probably an even superior alternative to deal with the diffusive fluxes.

In summary, the developed GPU accelerated rDG method has a great potential to become a viable, attractive, competitive and ultimately superior DG method over the current state-of-the-art second-order finite volume methods.

The outline of the dissertation is organized as follows. In Chapter 2, the governing equations and nondimensionalization are briefly described. In Chapter 3, the spatial discretization of these equations by the rDG methods is discussed in more detail. The developed hyperbolic reconstructed discontinuous Galerkin formulation is presented in detail in Chapter 4. The temporal integration methods (both explicit and implicit) are discussed in Chapter 5. Chapter 6 deals with the parallel implementation strategy using OpenACC. Numerical results for a variety of compressible inviscid/viscous flow test cases and several model equation cases are presented in Chapter 7. Finally the conclusion and future work is discussed in Chapter 8.
Chapter 2

Governing Equations of Fluid Dynamics

In this chapter, the governing equations of the physical flow models used in this work are briefly described: the Navier-Stokes equations (§2.1) and the Euler equations (§2.2). The nondimensionalization of the governing equations are described in the last section.

2.1 Navier-Stokes Equations

The Navier-Stokes equations governing unsteady compressible viscous flows can be expressed as

$$\frac{\partial U}{\partial t} + \frac{\partial F_k(U)}{\partial x_k} = \frac{\partial G_k(U)}{\partial x_k},$$

(2.1)

where the summation convention \((k = 1, 2, 3)\) has been used. The conservative variable vector \(U\) is defined by

$$U = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{pmatrix},$$

(2.2)

where \(\rho, p,\) and \(e\) denote the density, pressure, and specific total energy of the fluid, respectively, and \(u, v,\) and \(w\) are the velocity components of the flow in the coordinate direction \(x, y\) and \(z.\) The pressure can be computed from the equation of state

$$p = (\gamma - 1)\rho \left( e - \frac{1}{2}(u^2 + v^2 + w^2) \right),$$

(2.3)
which is valid for perfect gas, and the ratio of the specific heats $\gamma$ is assumed to be constant and equal to 1.4. Furthermore, the specific total enthalpy $h$ is defined as

$$h = e + \frac{p}{\rho}$$  \hspace{1cm} (2.4)

In Eq. 2.1, the advective (inviscid) flux tensor $\mathbf{F} = (\mathbf{F}_x, \mathbf{F}_y, \mathbf{F}_z)$ is defined by

$$\mathbf{F}_x = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho w \\ u(\rho e + p) \end{pmatrix}$$, \hspace{0.5cm} $$\mathbf{F}_y = \begin{pmatrix} \rho v \\ \rho u v \\ \rho u^2 + p \\ \rho w v \\ v(\rho e + p) \end{pmatrix}$$, \hspace{0.5cm} $$\mathbf{F}_z = \begin{pmatrix} \rho w \\ \rho u w \\ \rho w^2 + p \\ \rho w v \\ w(\rho e + p) \end{pmatrix}$$ \hspace{1cm} (2.5)

and the viscous flux tensor $\mathbf{G}$ is defined by

$$\mathbf{G}_x = \begin{pmatrix} 0 \\ \tau_{xx} \\ \tau_{xy} \\ \tau_{xz} \\ u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + q_x \end{pmatrix}$$

$$\mathbf{G}_y = \begin{pmatrix} 0 \\ \tau_{yx} \\ \tau_{yy} \\ \tau_{yz} \\ u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + q_y \end{pmatrix}$$ \hspace{1cm} (2.6)

$$\mathbf{G}_z = \begin{pmatrix} 0 \\ \tau_{zx} \\ \tau_{zy} \\ \tau_{zz} \\ u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + q_z \end{pmatrix}$$

where the viscous stress tensor $\tau$ is expressed as

$$\tau = \begin{pmatrix} \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \end{pmatrix}$$ \hspace{1cm} (2.7)
The Newtonian fluid with the Stokes hypothesis is valid under the current framework, since only air is considered. Thus $\tau$ is symmetric and the tensor is a linear function of the velocity gradients

$$\tau_{ij} = \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_k}{\partial x_k} \delta_{ij},$$  

where $\delta_{ij}$ is the Kronecker delta function, and $\mu$ represents the molecular viscosity coefficient (often referred to as dynamic viscosity coefficient as well), which can be determined through Sutherland’s law

$$\frac{\mu}{\mu_0} = \left( \frac{T}{T_0} \right)^{3/2} \frac{T_0 + S}{T + S},$$  

where $\mu_0$ denotes the viscosity coefficient at the reference temperature $T_0$, and $S$ is a constant which is assumed the value $S = 110K$. The temperature of the fluid $T$ is determined by

$$T = \frac{P}{\rho R},$$  

where $R$ denotes the universal gas constant for perfect gas.

The heat flux vector $q_j$, which is formulated according to Fourier’s law, is given by

$$q_j = -\lambda \frac{\partial T}{\partial x_j},$$  

where $\lambda$ is the thermal conductivity coefficient and expressed as

$$\lambda = \frac{\mu c_p}{Pr},$$  

where $c_p$ is the specific heat capacity at constant pressure and $Pr$ is the nondimensional laminar Prandtl number, which is taken as 0.7 for air.

### 2.2 Euler Equations

If the effect of viscosity and thermal conduction are neglected in Eq. 2.1, we arrived at the Euler equations expressed as below, which govern unsteady compressible inviscid flows

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}_k(\mathbf{U})}{\partial x_k} = 0.$$  

17
2.3 Nondimensionalization

The governing equations are often put into the nondimensional form. The advantage in doing so is that the characteristic parameters such as Mach number, Reynolds number, and Prandtl number can be varied independently. Also, by nondimensionalizing the governing equations, the flow variables are “normalized”, so that their values fall between certain prescribed limits such as 0 and 1. Many different nondimensionalizing procedures are possible. In this work, we use the following four reference variables: length, density, velocity and temperature. The choice of each reference variable is summarized in Table 2.1.

Table 2.1: Reference variables for nondimensionalization of the governing equations.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length $L_{ref}$</td>
<td>Problem dependent (cylinder diameter, plate length, etc) $d, l$</td>
</tr>
<tr>
<td>Density $\rho_{ref}$</td>
<td>Freestream density $\rho_\infty$</td>
</tr>
<tr>
<td>Velocity $V_{ref}$</td>
<td>Freestream speed of sound $a_\infty$</td>
</tr>
<tr>
<td>Temperature $T_{ref}$</td>
<td>Freestream temperature $T_\infty$</td>
</tr>
</tbody>
</table>

The nondimensional variables are denoted by an overbar

$$
\bar{L} = \frac{L}{L_{ref}}, \quad \bar{\rho} = \frac{\rho}{\rho_\infty}, \quad \bar{u} = \frac{u}{a_\infty}, \quad \bar{v} = \frac{v}{a_\infty}, \quad \bar{w} = \frac{w}{a_\infty}, \quad \bar{T} = \frac{T}{T_\infty},
$$

and accordingly, the derived normalized variables are expressed in the following manner

$$
\bar{p} = \frac{p}{\rho_\infty a_\infty^2}, \quad \bar{h} = \frac{h}{a_\infty^2}, \quad \bar{c}_p = \frac{c_p}{a_\infty^2/T_\infty}, \quad \bar{\mu} = \frac{\mu}{\rho_\infty a_\infty}, \quad \bar{\rho} = 1, \frac{\rho}{\gamma}.
$$

It is also trivial to derive the nondimensional equation of state as

$$
\bar{p} = \frac{1}{\gamma \bar{\rho} \bar{T}}.
$$

The freestream Mach number $M_\infty$ is defined as

$$
M_\infty = \frac{V_\infty}{a_\infty}.
$$
The freestream Reynolds number $Re_\infty$ is determined as

$$Re_\infty = \frac{\rho_\infty a_\infty L_{ref}}{\mu_\infty}. \quad (2.17)$$

The Prandtl number is written as

$$Pr = \frac{\mu_\infty c_p}{\lambda}. \quad (2.18)$$

In the normalized governing equations, the nondimensional viscous stress tensor is

$$\tau_{ij} = \bar{\mu} \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} - \frac{2}{3} \frac{\partial \bar{u}_k}{\partial x_k} \delta_{ij}\right), \quad (2.19)$$

and the nondimensional heat flux $\bar{q}_j$ vector is

$$\bar{q}_j = -\bar{\mu} \bar{c}_p \frac{1}{Pr} \frac{\partial \bar{T}}{\partial x_j}. \quad (2.20)$$

The nondimensional molecular viscosity coefficient $\bar{\mu}$ is computed with the dimensionless Sutherland’s law

$$\bar{\mu} = \frac{M_\infty}{Re_\infty} T_\infty^\frac{\gamma}{2} \frac{1 + S/T_\infty}{T + S/T_\infty}. \quad (2.21)$$

For the installation of a specific flow problem, the nondimensional input parameters include two fixed-value quantities $\bar{\rho}_\infty = 1.0$ and $a_\infty = 1.0$, and five user-adjustable quantities: $M_\infty$, angle of attack $\alpha$, yaw angle $\beta$, $Re_\infty$ and $Pr$. With these inputs, a uniform flow field is prescribed for a steady-state problem at the initialization stage and the corresponding conservative variables are

$$\bar{\rho}_\infty = 1.0, \quad (2.22)$$

$$\bar{\rho}u_\infty = M_\infty \cos \alpha \cos \beta, \quad (2.23)$$

$$\bar{\rho}v_\infty = M_\infty \cos \alpha \sin \beta, \quad (2.24)$$

$$\bar{\rho}w_\infty = M_\infty \sin \alpha, \quad (2.25)$$

$$\bar{\rho}e_\infty = \frac{1}{\gamma(\gamma - 1)} + \frac{1}{2} M_\infty^2. \quad (2.26)$$

The other derived dimensionless variables are

$$\bar{\rho}_\infty = \frac{1}{\gamma}, \quad (2.27)$$
\[
\bar{\mu}_\infty = \frac{M_\infty}{Re_\infty},
\]
\[
\bar{c}_p = \frac{1}{\gamma - 1},
\]
\[
\bar{\lambda} = \bar{\mu} \frac{1}{Pr} \frac{1}{\gamma - 1}.
\]

From the next chapter, all variables and equations that appear in the text are assumed to be in the nondimensional system and therefore the overbar sign will be dropped for simplicity.
Chapter 3

Reconstructed Discontinuous Galerkin Spatial Discretization

In the present work, the governing equations discretized in space by using the discontinuous Galerkin/ reconstructed discontinuous Galerkin method. The outline of this chapter is organized in the following. In section 3.1, the discontinuous Galerkin discretization for the Navier-Stokes equations is described in detail. The reconstruction approach for reconstructed discontinuous Galerkin methods are introduced in section 3.2. In section 3.3, the numerical flux scheme are shown. The finite element integration and the quadrature rules for several grid topologies are stated in section 3.4. Finally, the boundary conditions are described in section 3.5.

3.1 Discontinuous Galerkin Methods

3.1.1 Weak Formulation

The Navier-Stokes equations Eq. 2.1 are discretized using a discontinuous Galerkin finite element formulation. To formulate the discontinuous Galerkin method, we first introduce the following weak formulation, which is obtained by multiplying the above conservation law by a test function \( W \), integrating over the domain \( \Omega \), and then performing an integration by parts,

\[
\int_{\Omega} \frac{\partial U}{\partial t} W \, d\Omega + \int_{\Gamma} F_k n_k \, d\Gamma - \int_{\Omega} F_k \frac{\partial W}{\partial x_k} \, d\Omega = \int_{\Gamma} G_k n_k \, d\Gamma - \int_{\Omega} G_k \frac{\partial W}{\partial x_k} \, d\Omega, \quad (3.1)
\]

where \( \Gamma(= \partial \Omega) \) denotes the boundary of \( \Omega \), and \( n_j \) the unit outward normal vector to the boundary. We assume that the domain \( \Omega \) is subdivided into a collection of non-overlapping
arbitrary elements $\Omega_e$. Then we introduce the following broken Sobolev space $V_h^p$:

$$V_h^p = \{ v_h \in [L^2(\Omega)]^m : v_h|_{\Omega_e} \in [V_p^m] \forall \Omega_e \in \Omega \},$$

which consists of discontinuous vector-values polynomial functions of degree $p$, and where $m$ is the dimension of the unknown vector and

$$V_p^m = \text{span} \left\{ \prod x_i^{\alpha_i} : 0 \leq \alpha_i \leq p, 0 \leq i \leq d \right\},$$

where $\alpha$ denotes a multi-index and $d$ is the dimension of space. Then, we can obtain the following semi-discrete form by applying weak formulation on each element $\Omega_e$, find $U \in V_h^p$ such as

$$\frac{d}{dt} \int_{\Omega_e} U_h W_h \, d\Omega + \int_{\Gamma_e} F_k(U_h) n_k W_h \, d\Gamma - \int_{\Omega_e} F_k(U_h) \frac{\partial W_h}{\partial x_k} \, d\Omega$$

$$= -\int_{\Omega_e} G_k(U_h) n_k W_h \, d\Gamma - \int_{\Omega_e} G_k(U_h) \frac{\partial W_h}{\partial x_k} \, d\Omega, \quad \forall W_h \in V_h^p;$$

where $U_h$ and $W_h$ represent the finite element approximations to the analytical solution $U$ and the test function $W$ respectively, and they are approximated by a piecewise polynomial function of degrees $p$, which are discontinuous between the cell interfaces. Assume that $B$ is the basis of polynomial function of degrees $p$, this is then equivalent to the following system of $N$ equations,

$$\frac{d}{dt} \int_{\Omega_e} U_h B_i \, d\Omega + \int_{\Gamma_e} F_k(U_h) n_k B_i \, d\Gamma - \int_{\Omega_e} F_k(U_h) \frac{\partial B_i}{\partial x_k} \, d\Omega$$

$$= -\int_{\Gamma_e} G_k(U_h) n_k B_i \, d\Gamma - \int_{\Omega_e} G_k(U_h) \frac{\partial B_i}{\partial x_k} \, d\Omega, \quad 1 \leq i \leq N;$$

where $N$ is the dimension of the polynomial space. Since the numerical solution $U_h$ is discontinuous between element interfaces, the interface fluxes are not uniquely defined. The flux function $F_k(U_h) n_k$ appearing in the second terms of Eq. 3.5 is replaced by a numerical Riemann flux function $H_k(U_h^L, U_h^R, n_k)$ where $U_h^L$ and $U_h^R$ are the conservative state vectors at the left and right side of the element boundary. This scheme is called discontinuous Galerkin method of degree $p$, or in short notation DG(P) method. By simply increasing the degree $p$ of the polynomials, the DG methods of corresponding higher order are obtained.
3.1.2 Basis Functions

In the traditional DG method, numerical polynomial solutions $U_h$ in each element are represented using either standard Lagrange finite element or hierarchical node-based basis as follows:

$$U_h = \sum_{i=1}^{N} U_i B_i(x).$$  \hfill (3.6)

As a result, the unknowns to be solved are the variables at the nodes $U_i$, as illustrated in Fig. 3.1 for linear and quadratic polynomial approximations, where polynomial solutions are dependent on the shape of elements. For example, for a linear polynomial approximation in 2D as shown in Fig. 3.1, a linear polynomial is used for triangular elements and the unknowns to be solved are the variables at the three vertices and a bi-linear polynomial is used for quadrilateral elements and the unknowns to be solved are the variables at the four vertices. However, the numerical polynomial solutions $U$ can be expressed in other forms as well. In our work, the numerical polynomial solutions are represented using a Taylor series expansion at the centroid of the cell. In order to present the main ideas, let us consider $P_2$ approximation in 3D, where numerical solutions on each cell are approximated using a quadratic polynomial. If we do a Taylor series expansion at the centroid of the cell, the quadratic polynomial solutions can be expressed in other forms as well. In our work, the numerical polynomial solutions are represented using a Taylor series expansion at the centroid of the cell. In order to present the main ideas, let us consider $P_2$ approximation in 3D, where numerical solutions on each cell are approximated using a quadratic polynomial. If we do a Taylor series expansion at the centroid of the cell, the quadratic polynomial solutions can be expressed in other forms as well.
expressed as follows:

\( U_h = U_c + \frac{\partial U}{\partial x} (x - x_c) + \frac{\partial U}{\partial y} (y - y_c) + \frac{\partial U}{\partial z} (z - z_c) \)

\[ + \frac{\partial^2 U}{\partial x^2} (x - x_c)^2 + \frac{\partial^2 U}{\partial y^2} (y - y_c)^2 + \frac{\partial^2 U}{\partial z^2} (z - z_c)^2 \]

\[ + \frac{\partial^2 U}{\partial x \partial y} (x - x_c)(y - y_c) + \frac{\partial^2 U}{\partial x \partial z} (x - x_c)(z - z_c) + \frac{\partial^2 U}{\partial y \partial z} (y - y_c)(z - z_c) \]

(3.7)

where \( \bar{U} \) is the mean value of \( U \) in this cell. The unknowns to be solved in this formulation are the cell-averaged variables and their derivatives at the center of the cells, regardless of element shapes, as shown in Fig. 3.2. In this case, the dimension of the polynomial space is ten and the
Figure 3.2: Representation of polynomial solutions using a Taylor series expansion.

ten basis functions are

\[ B_1 = 1, \quad B_2 = x - x_c, \quad B_3 = y - y_c, \quad B_4 = z - z_c, \]
\[ B_5 = \frac{(x - x_c)^2}{2} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(x - x_c)^2}{2} d\Omega, \]
\[ B_6 = \frac{(y - y_c)^2}{2} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(y - y_c)^2}{2} d\Omega, \]
\[ B_7 = \frac{(z - z_c)^2}{2} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(z - z_c)^2}{2} d\Omega, \quad (3.9) \]
\[ B_8 = (x - x_c)(y - y_c) - \frac{1}{\Omega_e} \int_{\Omega_e} (x - x_c)(y - y_c) d\Omega, \]
\[ B_9 = (x - x_c)(z - z_c) - \frac{1}{\Omega_e} \int_{\Omega_e} (x - x_c)(z - z_c) d\Omega, \]
\[ B_{10} = (y - y_c)(z - z_c) - \frac{1}{\Omega_e} \int_{\Omega_e} (y - y_c)(z - z_c) d\Omega, \]

and the discontinuous Galerkin formulation Eq. 3.5 then leads to the following ten equations

\[ \frac{d}{dt} \int_{\Omega_e} \dot{U} \, d\Omega + \int_{\Gamma_e} F_k(U_h) n_k \, d\Gamma = \int_{\Gamma_e} G_k(U_h) n_k \, d\Gamma, \quad i = 1. \quad (3.10) \]
\[
\sum_{j=2}^{10} \int_{\Omega_e} B_i B_j \, d\Omega \frac{d}{dt} \left( \begin{array}{c} \frac{\partial U}{\partial x} |_c \\ \frac{\partial U}{\partial y} |_c \\ \frac{\partial U}{\partial z} |_c \\ \frac{\partial^2 U}{\partial x^2} |_c \\ \frac{\partial^2 U}{\partial y^2} |_c \\ \frac{\partial^2 U}{\partial z^2} |_c \\ \frac{\partial U}{\partial x \partial y} |_c \\ \frac{\partial U}{\partial x \partial z} |_c \\ \frac{\partial U}{\partial y \partial z} |_c \end{array} \right) + \int_{\Gamma_e} F_k(U_h) n_k B_i \, d\Gamma - \int_{\Omega_e} F_k(U_h) \frac{\partial B_i}{\partial x_k} \, d\Omega \quad (3.11)
\]

\[
= \int_{\Gamma_e} G_k(U_h) n_k B_i \, d\Gamma - \int_{\Omega_e} G_k(U_h) \frac{\partial B_i}{\partial x_k} \, d\Omega, \quad 2 \leq i \leq 10.
\]

Note that in this formulation, the equations for the cell-averaged variables are decoupled from the equations for their derivatives due to the judicious choice of the basis functions in our formulation and the fact

\[
\int_{\Omega_e} B_i B_j \, d\Omega = 0, \quad 2 \leq i \leq 10. \quad (3.12)
\]

Using this formulation, the similarity and difference between DG and FV methods become clear, and the advantage of the DG methods is especially evident in comparison with the FV methods. In fact, the discretized governing equations for cell-averaged variables Eq. 3.10 and the assumption of a polynomial solution on each cell Eq. 3.7 are exactly the same for both methods. In other words, this DG method provides a unified formulation, where the existing finite volume methods can be recovered virtually.

In the implementation of this DG method, the basis functions are actually normalized in
order to improve the conditioning of the system matrix Eq. 3.11 as follows:

\[
\begin{align*}
\tilde{B}_1 &= 1, \\
\tilde{B}_2 &= \frac{x - x_c}{\Delta x}, \\
\tilde{B}_3 &= \frac{y - y_c}{\Delta y}, \\
\tilde{B}_4 &= \frac{z - z_c}{\Delta z}, \\
\tilde{B}_5 &= \frac{(x - x_c)^2}{2\Delta x^2} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(x - x_c)^2}{2\Delta x^2} \, d\Omega, \\
\tilde{B}_6 &= \frac{(y - y_c)^2}{2\Delta y^2} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(y - y_c)^2}{2\Delta y^2} \, d\Omega, \\
\tilde{B}_7 &= \frac{(z - z_c)^2}{2\Delta z^2} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(z - z_c)^2}{2\Delta z^2} \, d\Omega, \\
\tilde{B}_8 &= \frac{(x - x_c)(y - y_c)}{\Delta x \Delta y} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(x - x_c)(y - y_c)}{\Delta x \Delta y} \, d\Omega, \\
\tilde{B}_9 &= \frac{(x - x_c)(z - z_c)}{\Delta x \Delta z} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(x - x_c)(z - z_c)}{\Delta x \Delta z} \, d\Omega, \\
\tilde{B}_{10} &= \frac{(y - y_c)(z - z_c)}{\Delta y \Delta z} - \frac{1}{\Omega_e} \int_{\Omega_e} \frac{(y - y_c)(z - z_c)}{\Delta y \Delta z} \, d\Omega,
\end{align*}
\] (3.13)

where \(\Delta x = 0.5(x_{\text{max}} - x_{\text{min}})\), \(\Delta y = 0.5(y_{\text{max}} - y_{\text{min}})\), \(\Delta z = 0.5(z_{\text{max}} - z_{\text{min}})\) and \(x_{\text{max}}, y_{\text{max}}, z_{\text{max}}\) and \(x_{\text{min}}, y_{\text{min}}, z_{\text{min}}\) are the maximum and minimum coordinates in the cell \(\Omega_e\) in \(x, y\) and \(z\) directions, respectively. Accordingly, the scaled first and second derivatives of the unknown variables are denoted as

\[
\begin{align*}
U_{x,c} &= \frac{\partial U}{\partial x} \bigg|_c \Delta x, \\
U_{y,c} &= \frac{\partial U}{\partial y} \bigg|_c \Delta y, \\
U_{z,c} &= \frac{\partial U}{\partial z} \bigg|_c \Delta z, \\
U_{xx,c} &= \frac{\partial^2 U}{\partial x^2} \bigg|_c \Delta x^2, \\
U_{yy,c} &= \frac{\partial^2 U}{\partial y^2} \bigg|_c \Delta y^2, \\
U_{zz,c} &= \frac{\partial^2 U}{\partial z^2} \bigg|_c \Delta z^2, \\
U_{xy,c} &= \frac{\partial^2 U}{\partial x \partial y} \bigg|_c \Delta x \Delta y, \\
U_{xz,c} &= \frac{\partial^2 U}{\partial x \partial z} \bigg|_c \Delta x \Delta z, \\
U_{yz,c} &= \frac{\partial^2 U}{\partial y \partial z} \bigg|_c \Delta y \Delta z.
\end{align*}
\] (3.14)

Finally, the quadratic polynomial solutions can be rewritten as

\[
\bar{U}_h = \tilde{U} + U_{x,c} \tilde{B}_2 + U_{y,c} \tilde{B}_3 + U_{z,c} \tilde{B}_4 + U_{xx,c} \tilde{B}_5 + U_{yy,c} \tilde{B}_6 + U_{zz,c} \tilde{B}_7 + U_{xy,c} \tilde{B}_8 + U_{xz,c} \tilde{B}_9 + U_{yz,c} \tilde{B}_{10}.
\] (3.15)

This is especially helpful and important to remove the stiffness of the system matrix for higher-order DG approximations.

This formulation has a number of distinct, desirable, and attractive features and advantages in the context of DG methods. First, the same numerical polynomial solutions are used for any
shape of elements, which can be triangle, quadrilateral and polygon in 2D, and tetrahedron, pyramid, prism and hexahedron in 3D. Using this formulation, the DG method can be easily implemented on arbitrary meshes. The numerical method based on this formulation has the ability to compute 1D, 2D and 3D problems using the very same code, which greatly alleviates the need and pain for code maintenance and upgrade. Secondly, the cell-averaged variables and their derivatives are handily available in this formulation. This makes implementation of WENO limiter straightforward and efficient [75,88,129,130,131], that is required to eliminate non-physical oscillations in the vicinity of discontinuities. Thirdly, the basis functions are hierarchic. This greatly facilitates implementation of $p$-multigrid methods [87,90] and $p$-refinement. Lastly, the cell-averaged variable equations are decoupled from their derivative equations in this formulation, which makes development of fast, low-storage implicit methods possible.

### 3.2 Reconstructed Discontinuous Galerkin Methods

Compared with the high order finite volume methods, the discontinuous Galerkin methods suffers from more computational costs and storage requirements. To be more specific, DG methods require more degrees of freedom, an additional domain integration, and more Gauss quadrature points for the boundary integration. On the one hand, the reconstruction methods that the FV methods use to achieve higher-order accuracy are relatively inexpensive but less accurate and robust. One the other hand, the DG methods that can be viewed as a different way to extend a FV method to higher orders are accurate and robust but costly. It is only natural and tempting to combine the efficiency of the reconstruction methods and the accuracy of the DG methods.

This idea was originally introduced by Dumbser et al. in the frame of $P_nP_m$ scheme [39,40,41], termed rDG($P_nP_m$) in this work, where $P_n$ indicates that a piecewise polynomial of degree of $n$ is used to represent a DG solution, and $P_m$ represents a reconstructed polynomial solution of degree of $m$ ($n \leq m$) that is used to compute the fluxes and source terms (if any). Therefore, Eq. 3.5 would be rewritten as

$$\frac{d}{dt} \int_{\Omega} U_h \, d\Omega + \int_{\Gamma_e} F_k(U_h^R) n_k B_i \, d\Gamma - \int_{\Omega_e} F_k(U_h^R) \frac{\partial B_i}{\partial x_k} \, d\Omega$$

$$= \int_{\Gamma_e} G_k(U_h^R) n_k B_i \, d\Gamma - \int_{\Omega_e} G_k(U_h^R) \frac{\partial B_i}{\partial x_k} \, d\Omega, \quad 1 \leq i \leq N, \tag{3.16}$$

where $U_h$ is the underlying $P_n$ DG solution and $U_h^R$ represents higher order $P_m$ reconstructed solutions.
The beauty of rDG($P_n P_m$) schemes is that they provide a unified formulation for both the FV and DG methods, and contain both the classical finite volume and standard discontinuous Galerkin methods as two special cases of rDG($P_n P_m$) schemes, and thus allow for a direct efficiency comparison. When $n = 0$, i.e. a piecewise constant polynomial is used to represent a numerical solution, rDG($P_0 P_m$) is nothing but the classical high order finite volume schemes, where a polynomial solution of degree $m$ ($m \leq 1$) is reconstructed from a piecewise constant solution. When $m = n$, the reconstruction reduces to the identity operator, and rDG($P_n P_m$) scheme yields a standard DG method. Clearly, an accurate and efficient reconstruction is the key ingredient in extending the underlying DG method to higher order accuracy. Although the discussion in this work is mainly focused on the rDG($P_1 P_2$), its extension to arbitrary order rDG methods is straightforward.

### 3.2.1 Least-Squares Reconstruction

In the case of DG($P_1$) method, a linear polynomial solution $U_i$ in any cell $i$ is

$$U_i = \hat{U}_i + U_{x,i} \hat{B}_2 + U_{y,i} \hat{B}_3 + U_{z,i} \hat{B}_4.$$  

(3.17)

Using this underlying linear polynomial DG solution in the neighboring cells, one can reconstruct a quadratic polynomial solution $U_i^R$ as follows:

$$U_i = \hat{U}_i^R + U_{x,i} \hat{B}_2 + U_{y,i} \hat{B}_3 + U_{z,i} \hat{B}_4 + U_{xx,i} \hat{B}_5 + U_{yy,i} \hat{B}_6 + U_{zz,i} \hat{B}_7 + U_{xy,i} \hat{B}_8 + U_{xz,i} \hat{B}_9 + U_{yz,i} \hat{B}_{10}.$$  

(3.18)

In order to maintain the compactness of the DG methods, the reconstruction is required to involve only von Neumann neighborhood, i.e., the adjacent cells that share a face with the cell $i$ under consideration, as shown in Fig. 3.3. There are ten degrees of freedom, and therefore ten unknowns must be determined. The first four unknowns can be trivially obtained, by requiring the consistency of the rDG method with the underlying DG method: (1) The reconstruction scheme must be conservative, and (2) The values of the reconstructed first derivatives are equal to the ones of the first derivatives of the underlying DG solution at the centroid $i$. Due to the judicious choice of Taylor basis in our DG formulation, these four degrees of freedom simply coincide with the ones from the underlying DG solution, i.e.,

$$\hat{U}_i^R = \hat{U}_i, \quad U_{x,i}^R = U_{x,i}, \quad U_{y,i}^R = U_{y,i}, \quad U_{z,i}^R = U_{z,i}.$$  

(3.19)

As a result, only six second derivatives need to be determined. This can be accomplished by
following Cheng’s work [24]. Consider a neighboring cell \( j \), one can denote \( \tilde{B}_j \) as the basis functions evaluated at the center of cell \( j \), that is,

\[
\begin{align*}
\tilde{B}_1^j &= 1, \\
\tilde{B}_2^j &= \frac{x_j - x_i}{\Delta x_i}, \\
\tilde{B}_3^j &= \frac{y_j - y_i}{\Delta y_i}, \\
\tilde{B}_4^j &= \frac{z_j - z_i}{\Delta z_i}, \\
\tilde{B}_5^j &= \frac{(x_j - x_i)^2}{2\Delta x_i^2} - \frac{1}{\Omega_i} \int_{\Omega_i} \frac{(x - x_i)^2}{2\Delta x_i^2} d\Omega, \\
\tilde{B}_6^j &= \frac{(y_j - y_i)^2}{2\Delta y_i^2} - \frac{1}{\Omega_i} \int_{\Omega_i} \frac{(y - y_i)^2}{2\Delta y_i^2} d\Omega, \\
\tilde{B}_7^j &= \frac{(z_j - z_i)^2}{2\Delta z_i^2} - \frac{1}{\Omega_i} \int_{\Omega_i} \frac{(z - z_i)^2}{2\Delta z_i^2} d\Omega, \\
\tilde{B}_8^j &= \frac{(x_j - x_i)(y_j - y_i)}{\Delta x_i \Delta y_i} - \frac{1}{\Omega_i} \int_{\Omega_i} \frac{(x - x_i)(y - y_i)}{\Delta x_i \Delta y_i} d\Omega, \\
\tilde{B}_9^j &= \frac{(x_j - x_i)(z_j - z_i)}{\Delta x_i \Delta z_i} - \frac{1}{\Omega_i} \int_{\Omega_i} \frac{(x - x_i)(z - z_i)}{\Delta x_i \Delta z_i} d\Omega, \\
\tilde{B}_{10}^j &= \frac{(y_j - y_i)(z_j - z_i)}{\Delta y_i \Delta z_i} - \frac{1}{\Omega_i} \int_{\Omega_i} \frac{(y - y_i)(z - z_i)}{\Delta y_i \Delta z_i} d\Omega.
\end{align*}
\]  

Thus, one would require

\[
\begin{align*}
\tilde{\mathbf{U}}_j &= \frac{1}{\Omega_j} \int_{\Omega_j} (\tilde{\mathbf{U}}_i^R + \mathbf{U}_{x,i}^R \tilde{B}_2 + \mathbf{U}_{y,i}^R \tilde{B}_3 + \mathbf{U}_{z,i}^R \tilde{B}_4 + \mathbf{U}_{xx,i}^R \tilde{B}_5 + \mathbf{U}_{yy,i}^R \tilde{B}_6 + \mathbf{U}_{zz,i}^R \tilde{B}_7 + \mathbf{U}_{xy,i}^R \tilde{B}_8 + \mathbf{U}_{yx,i}^R \tilde{B}_9 + \mathbf{U}_{yz,i}^R \tilde{B}_{10}) d\Omega, \\
\frac{\partial \mathbf{U}}{\partial x} |_{j} &= \mathbf{U}_{x,i}^R \frac{1}{\Delta x_i} + \mathbf{U}_{xx,i}^R \frac{\tilde{B}_2^j}{\Delta x_i} + \mathbf{U}_{xy,i}^R \frac{\tilde{B}_3^j}{\Delta x_i} + \mathbf{U}_{xz,i}^R \frac{\tilde{B}_4^j}{\Delta x_i}, \\
\frac{\partial \mathbf{U}}{\partial y} |_{j} &= \mathbf{U}_{y,i}^R \frac{1}{\Delta y_i} + \mathbf{U}_{yy,i}^R \frac{\tilde{B}_3^j}{\Delta y_i} + \mathbf{U}_{yx,i}^R \frac{\tilde{B}_2^j}{\Delta y_i} + \mathbf{U}_{yz,i}^R \frac{\tilde{B}_4^j}{\Delta y_i}, \\
\frac{\partial \mathbf{U}}{\partial z} |_{j} &= \mathbf{U}_{z,i}^R \frac{1}{\Delta z_i} + \mathbf{U}_{zz,i}^R \frac{\tilde{B}_4^j}{\Delta z_i} + \mathbf{U}_{xz,i}^R \frac{\tilde{B}_2^j}{\Delta z_i} + \mathbf{U}_{yz,i}^R \frac{\tilde{B}_3^j}{\Delta z_i}.
\end{align*}
\]
For simplification, the superscript $j^*$ would serve as the integral average operator on the neighboring cell $j$, i.e.,

$$\tilde{B}_k^{j^*} = \frac{1}{\Omega_j} \int_{\Omega_j} \tilde{B}_k d\Omega, \quad k = 1 \ldots N. \quad (3.22)$$

Finally, this group of equations can be written in a matrix form as follows:

$$\begin{pmatrix}
\tilde{B}_5^{j^*} & \tilde{B}_6^{j^*} & \tilde{B}_7^{j^*} & \tilde{B}_8^{j^*} & \tilde{B}_9^{j^*} & \tilde{B}_{10}^{j^*} \\
\tilde{B}_2^j & 0 & 0 & \tilde{B}_3^j & \tilde{B}_4^j & 0 \\
0 & \tilde{B}_3^j & 0 & \tilde{B}_2^j & 0 & \tilde{B}_4^j \\
0 & 0 & \tilde{B}_4^j & 0 & \tilde{B}_2^j & \tilde{B}_3^j
\end{pmatrix}
\begin{pmatrix}
U_{xx,i}^R \\
U_{yy,i}^R \\
U_{zz,i}^R \\
U_{xy,i}^R \\
U_{xz,i}^R \\
U_{yz,i}^R
\end{pmatrix} =
\begin{pmatrix}
R_1 \\
R_2 \\
R_3 \\
R_4
\end{pmatrix}, \quad (3.23)$$

where

$$\begin{pmatrix}
R_1 \\
R_2 \\
R_3 \\
R_4
\end{pmatrix} =
\begin{pmatrix}
\frac{\Delta x_j}{\Delta x_i} \hat{U}_{x,j} - \hat{U}_{x,i} \\
\frac{\Delta y_j}{\Delta y_i} \hat{U}_{y,j} - \hat{U}_{y,i} \\
\frac{\Delta z_j}{\Delta z_i} \hat{U}_{z,j} - \hat{U}_{z,i}
\end{pmatrix}.$$
Similar equations can be written for all the cells connected to the cell \( i \) with a common face, which leads to a non-square matrix. The numbers of the face-neighboring cells for a tetrahedron, a pyramid, a prism and a hexahedron are four, five, five and six, respectively. Consequently, the size of the resulting non-square matrix is \( 16 \times 6 \), \( 20 \times 6 \), \( 20 \times 6 \) and \( 24 \times 6 \), respectively. In the present work, this over-determined linear system of 16, or 20, or 24 equations for 6 unknowns is solved in the least-squares sense using both the normal equation approach and the QR decomposition to obtain the second derivatives of the reconstructed quadratic polynomial solution.

With the normal equation approach, a symmetric linear system of equations is yielded by pre-multiplying through by matrix transpose as follows

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\
a_{12} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\
a_{13} & a_{23} & a_{33} & a_{34} & a_{35} & a_{36} \\
a_{14} & a_{24} & a_{34} & a_{44} & a_{45} & a_{46} \\
a_{15} & a_{25} & a_{35} & a_{45} & a_{55} & a_{56} \\
a_{16} & a_{26} & a_{36} & a_{46} & a_{56} & a_{66}
\end{pmatrix}
\begin{pmatrix}
U_{xx,i}^R \\
U_{yy,i}^R \\
U_{zz,i}^R \\
U_{xy,i}^R \\
U_{xz,i}^R \\
U_{yz,i}^R
\end{pmatrix}
= 
\begin{pmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6
\end{pmatrix}, \quad (3.25)
\]
where

\[
\begin{align*}
    a_{11} &= \sum_{j}(\tilde{B}_5^{j*}\tilde{B}_5^{j*} + \tilde{B}_2^{j}\tilde{B}_2^{j}), \quad a_{12} = \sum_{j}(\tilde{B}_5^{j*}\tilde{B}_6^{j*}), \quad a_{13} = \sum_{j}(\tilde{B}_5^{j*}\tilde{B}_7^{j*}), \\
    a_{14} &= \sum_{j}(\tilde{B}_6^{j*}\tilde{B}_8^{j*} + \tilde{B}_2^{j}\tilde{B}_2^{j}), \quad a_{15} = \sum_{j}(\tilde{B}_6^{j*}\tilde{B}_9^{j}* + \tilde{B}_2^{j}\tilde{B}_4^{j}), \quad a_{16} = \sum_{j}(\tilde{B}_6^{j*}\tilde{B}_{10}^{j*}), \\
    a_{22} &= \sum_{j}(\tilde{B}_6^{j*}\tilde{B}_6^{j*} + \tilde{B}_3^{j}\tilde{B}_3^{j}), \quad a_{23} = \sum_{j}(\tilde{B}_7^{j*}\tilde{B}_7^{j*}), \quad a_{24} = \sum_{j}(\tilde{B}_6^{j*}\tilde{B}_8^{j*} + \tilde{B}_3^{j}\tilde{B}_4^{j}), \\
    a_{25} &= \sum_{j}(\tilde{B}_7^{j*}\tilde{B}_9^{j}), \quad a_{26} = \sum_{j}(\tilde{B}_7^{j*}\tilde{B}_{10}^{j} + \tilde{B}_4^{j}\tilde{B}_4^{j}), \quad a_{33} = \sum_{j}(\tilde{B}_7^{j*}\tilde{B}_7^{j*} + \tilde{B}_4^{j}\tilde{B}_4^{j}), \\
    a_{34} &= \sum_{j}(\tilde{B}_7^{j*}\tilde{B}_8^{j*}), \quad a_{35} = \sum_{j}(\tilde{B}_7^{j*}\tilde{B}_9^{j} + \tilde{B}_4^{j}\tilde{B}_2^{j}), \quad a_{36} = \sum_{j}(\tilde{B}_7^{j*}\tilde{B}_{10}^{j} + \tilde{B}_4^{j}\tilde{B}_3^{j}), \quad (3.26)
\end{align*}
\]

and

\[
\begin{align*}
    b_1 &= R_1\tilde{B}_5^{j*} + R_2\tilde{B}_2^{j}, \quad b_2 = R_1\tilde{B}_6^{j*} + R_3\tilde{B}_3^{j}, \\
    b_3 &= R_1\tilde{B}_7^{j*} + R_4\tilde{B}_4^{j}, \quad b_4 = R_1\tilde{B}_8^{j*} + R_2\tilde{B}_3^{j} + R_3\tilde{B}_3^{j}, \quad (3.27)
\end{align*}
\]

This linear system of $6 \times 6$ can be then trivially solved to obtain the second derivatives of the reconstructed quadratic polynomial solution. Furthermore, in implementation, the left-hand-side matrix in Eq. 3.25 can either be pre-computed for only once and stored at an extra expense of static memory requirement equal to $36 \times Nelem$ words, or computed during the loop in each time step with a dynamic memory allocation.

The QR decomposition (also called QR factorization) of a matrix is a decomposition of a matrix $A$ into a product $A = QR$ of an orthogonal matrix $Q$ and an upper triangular matrix $R$. QR decomposition is often used to solve the linear least squares system of equations, and is the basis for a particular eigenvalue algorithm, the QR algorithm. In our framework, the QR decomposition is implemented by calling the routines from the LAPACK (Linear Algebra PACKage) library. Compared with the normal equation approach, QR decomposition is a little more expensive in computing time, but is also observed to yield a slightly higher order of accuracy in a very few numerical experiments.

33
3.2.2 Variational Reconstruction

The $P_n P_m$ schemes require a reconstruction method which can extend the piecewise polynomial of order of $n$ to order of $m$. Most of work [37, 38, 105, 122] in this area is a continuation of the work of Barth and Frederickson [9] based on a least-squares (LS) reconstruction. However, the implementation of the least-squares method can suffer from two main disadvantages. First of all, the least-squares method may suffer from the linear instability on unstructured tetrahedral grids when the reconstruction stencils only involve von Neumann neighborhood, i.e., the adjacent face-neighboring cells [52]. Secondly, when it comes to higher order reconstruction, the stencils of the least-squares method must be extended. This extension of stencils may result in a more complicated data structure for unstructured grids. What’s more, the elements which are adjacent to the boundaries must be distinguished for choosing the stencils when implementing high-order least squares reconstruction.

Another reconstruction scheme has been developed recently based on variational formulation for finite volume method [147] and can be regarded as an extension of the compact finite difference schemes [70] to the unstructured grids. As a matter of fact, in §A, the prove is given to show that this variational reconstruction can recover to the compact finite difference scheme on 1D uniform grids. Different from the least squares approach, this variational reconstruction (VR) would solve an extreme value problem to obtain the higher moments of the solution. The problem minimizes the jumps of the values of the reconstructed polynomial solutions and their spatial derivatives at cell interfaces, and therefore maximizes smoothness of the reconstructed polynomial solutions. This reconstruction schemes has many attracting properties, including the $k$-exactness and the robustness. Unlike the LS method, the VR method is stable even on the tetrahedral grids, since its stencils are intrinsically the entire mesh. Unlike $rDG(P_n P_m)$ method based on the least square method, $rDG_{VR}(P_n P_m)$ can easily extend to higher order of $m$ due to the fact that it only require the compact data structure but has global stencil. Moreover, this would also make it easier for parallelization.

Variational reconstruction defines the global cost function as the summation of all the local cost function at each face. For simplification, one can only consider the contributions from the inner faces. Hence, the cost function can be defined as

$$I = \sum_{iface=1}^{Nface} I_{iface},$$

where $Nface$ is the number of cell interfaces. For an given face $\Gamma_{ij}$ separating the left element
and the right element \( j \), the local cost function is defined as

\[
I_{i,\text{face}} = \frac{1}{d_{ij}} \int_{\Gamma_{ij}} \sum_{p=0}^{k} \sum_{m=0}^{p} \sum_{n=0}^{m} (w_{m,n,p} \partial^p U \partial x^n \partial y^m \partial z^{p-m-n} d_{ij})^2 d\Gamma,
\]

where \( d_{ij} \) is the distance between the centroid of cell \( i \) and \( j \), \( w_{m,n,p} \) is the weight associated with certain derivatives, and \( \langle \cdot \rangle \) denotes the jump operator between the two side. This face integral can be computed exactly using Gaussian quadrature formulas with sufficient precision, whose details could be found later. The constitutive relations of the variational reconstruction are derived by minimizing the total cost function \( I \) with respect to the coefficients of the reconstruction polynomial. One can solve this problem by using Lagrange multipliers. This leads to a system of linear equations to be solved in each time step. Indeed, compared with LS reconstruction, the linear system that needs to be solved has much larger dimension due to the fact all the higher unknowns across the domain are coupled. Nevertheless, one do not need to solve this system completely in each time step. Instead, performing one iteration at each time step using some iterative linear solver, for instance, symmetric Gauss-Seidel (SGS) is implemented in this work, should be sufficient in practice.

### 3.2.3 WENO Reconstruction at \( P_2 \): WENO(P\(_1\)P\(_2\))

This linear reconstruction-based rDG(P\(_1\)P\(_2\)) method is able to achieve the designed third order of accuracy and significantly improve the accuracy of the underlying second-order DG method for solving the 2D compressible Euler equations on arbitrary grids \([92, 94, 98, 161]\). However, when used to solve the 3D compressible Euler equations on tetrahedral grids or prismatic grids, this rDG method suffers from the so-called linear instability, that is also observed in the second-order cell-centered finite volume methods, i.e., rDG(P\(_0\)P\(_1\)) \([51]\). This linear instability is attributed to the fact that the reconstruction stencils only involve the von Neumann neighborhood, i.e., adjacent face-neighboring cells \([51]\). The linear stability can be achieved using extended stencils, which will unfortunately sacrifice the compactness of the underlying DG methods. Furthermore, such a linear reconstruction-based DG method will suffer from non-linear instability, leading to non-physical oscillations in the vicinity of strong discontinuities for the compressible Euler equations. Alternatively, the ENO (Essentially Non-Oscillatory) scheme, WENO (Weighted Essentially Non-Oscillatory) scheme, and Hermite WENO scheme can be used to reconstruct a higher-order polynomial solution, which can not only enhance the order of accuracy of the underlying DG method but also achieve both linear and non-linear stability. In the present work, the reconstructed quadratic polynomial based on the Hermite WENO on cell \( i \) are a
convex combination of the least-squares reconstructed second derivatives at the cell itself and its face-neighboring cells,

\[
\frac{\partial^2 U}{\partial x_m \partial x_n} \bigg|_{\text{WENO}} = \sum_{k=1}^{1+N_{es}} w_k \frac{\partial^2 U}{\partial x_m \partial x_n} |_k ,
\]

where \( N_{es} \) denotes the number of its face-neighboring cells (\( N_{es} \) is 4 for tetrahedron, 5 for pyramid and prism, and 6 for hexahedron), and the normalized nonlinear weights \( w_k \) are computed as

\[
w_k = \frac{\tilde{w}_k}{1 + N_{es} \sum_{i=1} w_i} .
\]

The non-normalized nonlinear weights \( \tilde{w}_i \) are functions of the linear weights \( \lambda_i \) and the so-called oscillation indicator \( o_i \)

\[
\tilde{w}_k = \frac{\lambda_i}{(\varepsilon + o_i)^\gamma} ,
\]

where \( \varepsilon \) is a small positive number used to avoid division by zero, and \( \gamma \) an integer parameter to control how fast the non-linear weights decay for non-smooth stencils. The oscillation indicator \( o_k \) for the reconstructed second order polynomials is simply defined as

\[
o_k = \left( \frac{\partial^2 U}{\partial x_m \partial x_n} |_k \right)^2 \right)^{\frac{1}{2}} ,
\]

where the convention of summation over repeated indices \( m \) and \( n \) (\( m, n = 1, 2, 3 \)) are used.

Note that the least-squares reconstructed polynomial at the cell itself serves as the central stencil and the least-squares reconstructed polynomials on its face-neighboring cells act as biased stencils in this Hermite WENO reconstruction. This reconstructed quadratic polynomial solution is then used to compute the domain and boundary integrals of the underlying DG(P1) method in Eq. 3.5. The resulting DG method, termed a “reconstructed DG” method (rDG(P1P2) in short notation), is expected to have third order of accuracy at a moderate increase of computing costs in comparison to the underlying DG(P1) method. The extra costs are mainly due to the least-squares reconstruction, which is relatively cheap in comparison to the evaluation of fluxes, and an extra Gauss quadrature point, which is required to calculate both domain and boundary integrals on tetrahedral cells. In comparison to DG(P2), this represents a significant saving in terms of flux evaluations. Furthermore, the number of degrees of freedom is considerably reduced, which leads to a significant reduction in memory requirements, and
from which implicit methods will benefit tremendously. The cost analysis for the \( \text{rDG}(P_0P_1), \text{rDG}(P_1P_1) \) (DG(P_1)), \( \text{rDG}(P_1P_2) \) and \( \text{rDG}(P_2P_2) \) (DG(P_2)) on a tetrahedral cell is summarized in Table 3.1, where the memory requirement for storing only the implicit diagonal matrix is given as well, and which grows quadratically with the order of the DG methods. We would like to emphasize that the storage requirements for the implicit DG methods are extremely demanding, especially for higher-order DG methods. A similar summary of cost analysis on a hexahedral grid is presented in Table 3.2, where the numbers of Gauss quadrature points for both the domain and boundary integrals required by the \( \text{rDG}(P_1P_2) \) method are the same with the \( \text{rDG}(P_1P_1) \) method, and are a huge reduction in comparison with the \( \text{rDG}(P_2P_2) \) method. Note that this \( \text{rDG} \) method is not compact anymore, as the neighbor of neighbors are used in updating the solution. However, the stencils used in the reconstruction are compact, involving only von Neumann neighbors. Consequently, the resultant \( \text{rDG} \) method can be implemented in a compact manner. If not specially stated, “\( \text{rDG}(P_1P_2) \) method” refers to WENO(P_1P_2), and “\( \text{rDG}(P_1P_2) \) solution” refers to the WENO(P_1P_2) reconstructed quadratic polynomial solution in chapter 7. As demonstrated in chapter 7, this \( \text{rDG}(P_1P_2) \) method is able to achieve the designed third order of accuracy for inviscid smooth flows and maintain the desired linear stability.

Table 3.1: Cost analysis for different numerical methods on a tetrahedral grid.

<table>
<thead>
<tr>
<th></th>
<th>( \text{rDG}(P_0P_1) )</th>
<th>( \text{rDG}(P_1P_1) )</th>
<th>( \text{rDG}(P_1P_2) )</th>
<th>( \text{rDG}(P_2P_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of quadrature points</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>for boundary integrals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of quadrature points</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>for domain integrals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reconstruction</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Order of accuracy</td>
<td>( \mathcal{O}(h^2) )</td>
<td>( \mathcal{O}(h^2) )</td>
<td>( \mathcal{O}(h^3) )</td>
<td>( \mathcal{O}(h^3) )</td>
</tr>
<tr>
<td>Storage for implicit diagonal</td>
<td>25 words per element</td>
<td>400</td>
<td>400</td>
<td>2500</td>
</tr>
<tr>
<td>matrix</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.2.4 WENO Reconstruction at \( P_1 \): HWENO(P_1P_2)

Although the WENO(P_1P_2) method does not introduce any new oscillatory behavior for the reconstructed curvature terms (second derivatives) due to the WENO reconstruction, it cannot re-
Table 3.2: Cost analysis for different numerical methods on a hexahedral grid.

<table>
<thead>
<tr>
<th></th>
<th>rDG($P_0P_1$)</th>
<th>rDG($P_1P_1$)</th>
<th>rDG($P_1P_2$)</th>
<th>rDG($P_2P_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of quadrature points</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>for boundary integrals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of quadrature points</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>27</td>
</tr>
<tr>
<td>for domain integrals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reconstruction</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Order of accuracy</td>
<td>$O(h^2)$</td>
<td>$O(h^2)$</td>
<td>$O(h^3)$</td>
<td>$O(h^3)$</td>
</tr>
<tr>
<td>Storage for implicit diagonal</td>
<td>25 words per</td>
<td>400</td>
<td>400</td>
<td>2500</td>
</tr>
<tr>
<td>matrix</td>
<td>element</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

move inherent oscillations in the underlying DG($P_1$) solutions. Consequently, the WENO($P_1P_2$) method still suffers from the non-linear instability for flows with strong discontinuities. In order to eliminate non-physical oscillations in the vicinity of strong discontinuities and thus maintain the non-linear instability, the first derivatives need to be reconstructed using a WENO reconstruction. The resulting reconstructed discontinuous Galerkin method based on this Hierarchical WENO reconstruction is termed as HWENO($P_1P_2$) [99] in this work, where a hierarchical reconstruction (successively from high order to low order) strategy [158] is adopted.

The WENO reconstruction for the first derivatives is based on the reconstructed quadratic polynomial solutions of the flow variables for each cell in the grid. The stencils are only chosen in the von Neumann neighborhood. More precisely, for a tetrahedral cell $i$, the following four stencils ($i,j_1,j_2,j_3$), ($i,j_1,j_2,j_4$), ($i,j_1,j_3,j_4$) and ($i,j_2,j_3,j_4$), where $j_1$, $j_2$, $j_3$ and $j_4$ designate the four adjacent face-neighboring cells of the cell $i$ are chosen to construct a Lagrange polynomial such that

$$
\tilde{U}_j = \tilde{U}_i^R + U_{x,i}^R \tilde{B}_{j2}^* + U_{y,i}^R \tilde{B}_{j3}^* + U_{z,i}^R \tilde{B}_{j4}^* + U_{xx,i}^R \tilde{B}_j^{j*} + U_{yy,i}^R \tilde{B}_j^{j*} + U_{zz,i}^R \tilde{B}_j^{j*} + U_{xy,i}^R \tilde{B}_j^{j*} + U_{xz,i}^R \tilde{B}_j^{j*} + U_{yz,i}^R \tilde{B}_j^{j*},
$$

where $\tilde{U}_j$ refers to the cell-averaged value of the reconstructed polynomial solution at centroid of cell $j$. In addition, the following four stencils ($i,j_1$), ($i,j_2$), ($i,j_3$) and ($i,j_4$) are chosen to
construct a Hermite polynomial such that

\[
\frac{\partial U}{\partial x} = U^R_{x,i} \frac{1}{\Delta x_i} + U^R_{x,y,i} \frac{\tilde{B}_2}{\Delta x_i} + U^R_{x,z,i} \frac{\tilde{B}_3}{\Delta x_i},
\]

\[
\frac{\partial U}{\partial y} = U^R_{y,i} \frac{1}{\Delta y_i} + U^R_{y,z,i} \frac{\tilde{B}_2}{\Delta y_i} + U^R_{y,z,i} \frac{\tilde{B}_3}{\Delta y_i},
\]

\[
\frac{\partial U}{\partial z} = U^R_{z,i} \frac{1}{\Delta z_i} + U^R_{z,y,i} \frac{\tilde{B}_4}{\Delta z_i} + U^R_{z,y,i} \frac{\tilde{B}_3}{\Delta z_i}.
\]\({3.35}\)

These eight reconstructed gradients \((U^R_{x,i}, U^R_{y,i}, \text{and } U^R_{z,i})\) serving as the biased stencils and the gradient from the DG solution itself at cell \(i\) \((U_{x,i}, U_{y,i}, \text{and } U_{z,i})\) acting as the central stencil are used to modify the first derivatives based on the WENO reconstruction as a convex combination of these nine derivatives

\[
\frac{\partial U}{\partial x} \bigg|_{\text{WENO}} = \sum_{k=1}^{N_{\text{sten}}} w_k \frac{\partial U}{\partial x} \bigg|_{k},
\]\({3.36}\)

where, \(N_{\text{sten}}\) denotes the number of stencils. In general, \(N_{\text{sten}}\) is equal to \(C_{N_{es}}^3 + N_{es} + 1\) for an element (9 for a tetrahedron, 16 for a pyramid and prism, and 27 for a hexahedron). The normalized nonlinear weights \(w_k\) are computed as

\[
w_k = \frac{\tilde{w}_k}{\sum_{i=1}^{N_{\text{sten}}} \tilde{w}_i}.
\]\({3.37}\)

The non-normalized nonlinear weights \(\tilde{w}_i\) are functions of the linear weights \(\lambda_i\) and the so-called oscillation indicator \(o_i\)

\[
\tilde{w}_k = \frac{\lambda_i}{(\varepsilon + o_i)^\gamma},
\]\({3.38}\)

where \(\varepsilon\) is a small positive number used to avoid division by zero, and \(\gamma\) an integer parameter to control how fast the non-linear weights decay for non-smooth stencils. The oscillation indicator \(o_k\) for the reconstructed first order polynomials is simply defined as

\[
o_k = \left[ \left( \frac{\partial U}{\partial x} \bigg|_k \right)^2 \right]^{\frac{1}{2}},
\]\({3.39}\)
where the convention of summation over repeated indices $m$ ($m = 1, 2, 3$) is used. The present choice of stencils is symmetric, and compact, as the van Neumann neighbors are only involved in the reconstruction. This means that no additional data structure is required for our HWENO ($P_1P_2$) method. Note that this WENO reconstruction at $P_1$ is the extension of a HWENO limiter developed for the DG ($P_1$) by Luo et al. [88]. From the perspective of both computational cost and solution accuracy, the above WENO reconstruction at $P_1$ should only be used in the regions where strong discontinuities exist. This can be accomplished using the so-called discontinuity detectors, which are helpful to distinguish regions where solutions are smooth and discontinuous. The beauty of this WENO reconstruction is that in case that the reconstruction is mistakenly applied in the smooth cells, the uniform high-order accuracy can still be maintained, unlike the slope limiters, which, when applied near smooth extrema, will have a profoundly adverse impact on solution in the smooth region, leading to the loss of the original high-order accuracy. This remarkable feature of the WENO reconstruction in turn alleviates the burden on the discontinuity detectors, as no discontinuity detectors can really either in theory or in practice make a distinction between a stagnation point and a shock wave, as flow gradients near the stagnation point are even larger than the ones near the shock wave in some cases. The developed HWENO($P_1P_2$) scheme is assessed and validated to achieve the designed third order of accuracy for both internal and external smooth inviscid flow problems: one order of accuracy higher than the underlying DG($P_1$) method, as presented in Reference [99].

3.3 Numerical Flux

3.3.1 Inviscid Flux Scheme

In our work, the inviscid flux function is approximated using the HLLC approximate Riemann solver [142], which has been successfully used to compute compressible viscous and turbulent flows on both structured grids [17] and unstructured grids [85]. This HLLC scheme is found to have the following properties: (1) exact preservation of isolated contact and shear waves, (2) positivity-preserving of scalar quantity, (3) enforcement of entropy condition. In addition, the implementation of HLLC Riemann solver is easier and the computational cost is lower compared with some other available Riemann solvers.

The explicit form of the HLLC flux evaluated at the integration point of face $ij$ is defined
evaluated at the face integration point respectively as follows

\[ \mathbf{H}_{\text{HLLC}}(\mathbf{U}_l, \mathbf{U}_r, \mathbf{n}_{ij}) = \begin{cases} 
\mathbf{H}_l(\mathbf{U}_l) & \text{if } S_L > 0 \\
\mathbf{H}(\mathbf{U}_l^*) & \text{if } S_L \leq 0 < S_M \\
\mathbf{H}(\mathbf{U}_r^*) & \text{if } S_M \leq 0 \leq S_R \\
\mathbf{H}_r(\mathbf{U}_r) & \text{if } S_R < 0 
\end{cases}, \quad (3.40) \]

where the subscript \( l \) and \( r \) denote the elemental state vectors \( \mathbf{U}_i \) from cell \( i \) and \( \mathbf{U}_j \) from cell \( j \) evaluated at the face integration point respectively as follows

\[
\mathbf{U}_l = \begin{pmatrix} 
\rho_l \\
(pu)_l \\
(pv)_l \\
(pw)_l \\
(pe)_l 
\end{pmatrix} = \sum_{id}^{M\text{degr}} \begin{pmatrix} 
\tilde{B}_{id,1} \mathbf{U}_{id,1,l} \\
\tilde{B}_{id,1} \mathbf{U}_{id,2,l} \\
\tilde{B}_{id,1} \mathbf{U}_{id,3,l} \\
\tilde{B}_{id,1} \mathbf{U}_{id,4,l} \\
\tilde{B}_{id,1} \mathbf{U}_{id,5,l} 
\end{pmatrix}, \quad \mathbf{U}_r = \begin{pmatrix} 
\rho_r \\
(pu)_r \\
(pv)_r \\
(pw)_r \\
(pe)_r 
\end{pmatrix} = \sum_{id}^{M\text{degr}} \begin{pmatrix} 
\tilde{B}_{id,1} \mathbf{U}_{id,1,r} \\
\tilde{B}_{id,1} \mathbf{U}_{id,2,r} \\
\tilde{B}_{id,1} \mathbf{U}_{id,3,r} \\
\tilde{B}_{id,1} \mathbf{U}_{id,4,r} \\
\tilde{B}_{id,1} \mathbf{U}_{id,5,r} 
\end{pmatrix}, \quad (3.41) \]

where \( M\text{degr} \) is the number of DOFs for the polynomials (\( M\text{degr} = 1 \) for rDG(P_0P_0), \( M\text{degr} = 4 \) for rDG(P_0P_1) and rDG(P_1P_1), \( M\text{degr} = 10 \) for rDG(P_1P_2) and rDG(P_2P_2)).

In Eq. 3.40, the symbols with superscript * are defined by

\[
\mathbf{U}_l^* = \begin{pmatrix} 
\rho_l^* \\
(pu)_l^* \\
(pv)_l^* \\
(pw)_l^* \\
(pe)_l^* 
\end{pmatrix} = \Omega_l \begin{pmatrix} 
\rho_l(S_L - q_l) \\
(S_L - q_l)(pu)_l + (p^* - p_l)n_x \\
(S_L - q_l)(pv)_l + (p^* - p_l)n_y \\
(S_L - q_l)(pw)_l + (p^* - p_l)n_z \\
(S_L - q_l)(pe)_l - p_l q_l + p^* S_M 
\end{pmatrix}, \quad (3.42) \]

\[
\mathbf{U}_r^* = \begin{pmatrix} 
\rho_r^* \\
(pu)_r^* \\
(pv)_r^* \\
(pw)_r^* \\
(pe)_r^* 
\end{pmatrix} = \Omega_r \begin{pmatrix} 
\rho_r(S_R - q_r) \\
(S_R - q_r)(pu)_r + (p^* - p_r)n_x \\
(S_R - q_r)(pv)_r + (p^* - p_r)n_y \\
(S_R - q_r)(pw)_r + (p^* - p_r)n_z \\
(S_R - q_r)(pe)_r - p_r q_r + p^* S_M 
\end{pmatrix}, \quad (3.43) \]

\[
\mathbf{H}_l^* \equiv \mathbf{H}(\mathbf{U}_l^*) = \begin{pmatrix} 
\rho_l^* S_M \\
(pu)_l^* S_M + p^* n_x \\
(pv)_l^* S_M + p^* n_y \\
(pw)_l^* S_M + p^* n_z \\
((pe)_l^* + p^*) S_M 
\end{pmatrix}, \quad \mathbf{H}_r^* \equiv \mathbf{H}(\mathbf{U}_r^*) = \begin{pmatrix} 
\rho_r^* S_M \\
(pu)_r^* S_M + p^* n_x \\
(pv)_r^* S_M + p^* n_y \\
(pw)_r^* S_M + p^* n_z \\
((pe)_r^* + p^*) S_M 
\end{pmatrix}, \quad (3.44) \]

\[
\Omega_l \equiv (S_L - S_M)^{-1}, \quad \Omega_r \equiv (S_R - S_M)^{-1}, \quad (3.45) \]
\[ p^* = \rho_l (q_l - S_L) (q_l - S_M) + p_l = \rho_r (q_r - S_R) (q_r - S_M) + p_r, \]
\[ q_l \equiv u_l n_x + v_l n_y + w_l n_z, \quad q_r \equiv u_r n_x + v_r n_y + w_r n_z, \]
with \( (n_x, n_y, n_z)^T \) being the unit vector normal to face \( ij \). \( S_M \) is taken from Batten et al. [16]:
\[ S_M = \frac{\rho_r q_r (S_R - q_r) - \rho_l q_l (S_L - q_l) + p_l - p_r}{\rho_r (S_R - q_r) - \rho_l (S_L - q_l)}, \]
and \( S_L, S_R \) are taken from Einfeldt et al. [43]:
\[ S_L = \min \left[ \lambda_1(U_l), \lambda_1(U^{\text{Roe}}) \right], \quad S_R = \max \left[ \lambda_m(U^{\text{Roe}}), \lambda_m(U_r) \right], \]
where \( \lambda_1(U^{\text{Roe}}) \) and \( \lambda_m(U^{\text{Roe}}) \) are the smallest and largest eigenvalues of the Roe matrix [134].

### 3.3.2 Viscous Flux Scheme

In contrast to the various successful schemes to treat the inviscid flux, an optimal scheme to discretize the viscous flux is still under quest for the discontinuous Galerkin methods. Among the most remarkable are the first Bassi-Rebay (BR1) scheme that was introduced by Bassi and Rebay [11] in 1997 and the famous second Bassi-Rebay (BR2) scheme [13] that was introduced in 1999. The BR2 scheme was designed to overcome several deficiencies of the BR1 scheme, and received a wide acceptance in the DG community. In Bassi and Rebay’s 2005 publication [15], the Reynolds-Averaged Navier-Stokes (RANS) equations and \( k-\varepsilon \) turbulence model were first discretized in the discontinuous Galerkin space and integrated implicitly in time with the analytically derived approximate Jacobians. We have chosen to implement the famous second Bassi-Rebay scheme (BR2) [15] for the discretization of the viscous fluxes, as the BR2 scheme is the only one proposed in the literature to achieve optimal order of accuracy of compactness.

To apply the BR2 scheme, the numerical flux at the integration point of face \( ij \) is given as
\[ H_{\text{BR2}}^i = H_{\text{BR2}}^i (U_l, \nabla U_l + r_l, U_r, \nabla U_r + r_r), \]
where \( r \) is the so-called local lifting operator for interior faces, and defined by
\[ \int_{\Omega_i} r_i B_i \ d\Omega = \int_{\Gamma_{ij}} \frac{1}{2} (U_r - U_l) n_k B_i \ d\Gamma. \]
On the boundary faces the local lifting operator is defined by

\[ \int_{\Omega_i} \mathbf{r}_i B_i \, d\Omega = \int_{\Gamma_{ij}} (\mathbf{U}_b - \mathbf{U}_l)n_k B_i \, d\Gamma. \]  

(3.52)

The locality means that the integration is evaluated on face \( ij \) only.

The domain integral evaluated at the integration point of cell \( i \) is defined by

\[ G_n^{BR2}(\mathbf{U}_i, \nabla \mathbf{U}_i + \mathcal{R}_i) = G_n^{BR2}(\mathbf{U}_i, \nabla \mathbf{U}_i + \mathcal{R}_i), \]  

(3.53)

where \( \mathcal{R} \) is the so-called global lifting operator for elements, and defined by

\[ \int_{\Omega_i} \mathcal{R}_i B_i \, d\Omega = \int_{\partial \Omega_i} \frac{1}{2}(\mathbf{U}_b - \mathbf{U}_l)n_k B_i \, d\Gamma. \]  

(3.54)

The relation between the local and global lifting operators is given by

\[ \mathcal{R} = \sum_{j \neq i} \frac{r_{ij}}{r_{ij}} \]  

(3.55)

The BR2 scheme was successfully implemented in various studies in the DG community and many remarkable Navier-Stokes solvers based on have been developed recently. In 2006, Klaij et. al [66] introduced a space-time discontinuous Galerkin method for the compressible Navier-Stokes equations, in which the procedures to assemble the approximate Jacobians of the BR2 viscous flux function were described in detail. In 2008, Landmann [69] developed a parallel discontinuous Galerkin code for the Navier-Stokes and RANS equations on 2D structured/unstructured grids, in which differentiation of the BR2 viscous flux function were carried out with the aid of the algebraic package Maple. In 2010, Yasue et. al [159] developed an implicit discontinuous Galerkin method for RANS simulation, in which the inviscid flux was linearized using a spectral method while the linearization of BR2 viscous flux was accurately approximated. More recently, Crivellini et. al [35] introduced an implicit matrix-free discontinuous Galerkin solver, in which a matrix-free GMRES algorithm was extended into the discontinuous Galerkin space for viscous and turbulent aerodynamic simulations. All these works discussed above indicate that the key to a robust and efficient implicit DG Navier-Stokes solver is the quality of the linearized viscous terms. On the other side, a common disadvantage of these solvers is that they are required to devote large amount of CPU time to computing the viscous Jacobian matrix for each or every several Newton iterations, as the complexity of computation...
increases cubically with the order of the considered DG methods.

As a matter of fact, how to efficiently discretize the viscous flux using DG formulation has been a challenge for a long time. Another approach was taken by adopting the idea of first order hyperbolic system (FOHS) method to develop a hyperbolic DG/rDG method [81]. Further discussion of this new method would be given later, along with the comparison with some conventional DG diffusive flux method.

3.4 Numerical Integration

Numerical integration for the discontinuous Galerkin formulation is often carried out using the numerical quadrature formulae. In our study, we adopt the Gaussian integration quadrature formulae in discontinuous finite element calculations since they present the best accuracy for a given number of points. The objective of the effort discussed in this section is only limited to apply the numerical integration to the semi-discretized weak formulation of the governing equations as shown in Eq. 3.5, but not trying to provide the detailed procedures of numerical integration, as the principles and complete concepts can be referred to some well-accepted textbooks [58, 72].

![Figure 3.4: Transformation of an element in (x, y) physical space into a canonical element reference (ξ, η) space.](image)

Numerical integration is actually performed with respect to the reference domain. To begin with, our focus is on boundary calculations. For an isoparametric formulation, the nodes for numerical integration in the faces are mapped to a reference (or canonical) face, and all ensuing
calculations are conducted in the mapped face. An example is shown in Fig. 3.4, where a curvilinear quadrilateral face is mapped to its corresponding 2D reference face in the normalized coordinate system. The transformation between the face in physical space and the reference face is invertible, or in other words, the Jacobian of the transformation matrix is positive. The quadrilateral reference face is chosen to be a square, where the local coordinates \( \xi \) and \( \eta \) are normalized, \( \{ (\xi, \eta) \mid \xi, \eta \in [-1, +1] \} \).

If we treat the coordinate variables \( x, y \) and \( z \) themselves as functions on \( \Gamma \), then they can be approximated using the standard finite element shape functions in the form

\[
\begin{align*}
  x(\xi, \eta) &= \sum_{i} x_i \phi_i(\xi, \eta), \\
  y(\xi, \eta) &= \sum_{i} y_i \phi_i(\xi, \eta), \\
  z(\xi, \eta) &= \sum_{i} z_i \phi_i(\xi, \eta),
\end{align*}
\]

(3.56)

where \( (x_i, y_i, z_i) \) are the \( (x, y, z) \) coordinates of local nodal point \( i \) of face \( \Gamma_e \). \( N_f \) is equal to the number of the shape functions for the face. Note that by this transformation, the mapping can be constructed from every face in the physical space to the reference face using the shape functions. The 4-node and shape functions and derivatives for linear quadrilateral element and 8-node shape functions for curvilinear quadrilateral element are expressed in §B.2.2 and §B.2.3, respectively.

Note that the functions \( x, y \) and \( z \) are differentiable with respect to the local coordinates \( \xi \) and \( \eta \) in reference space:

\[
\begin{align*}
  dx(\xi, \eta) &= \frac{\partial x}{\partial \xi} d\xi + \frac{\partial x}{\partial \eta} d\eta = \sum_{i} x_i \frac{\partial \phi_i(\xi, \eta)}{\partial \xi} d\xi + \sum_{i} x_i \frac{\partial \phi_i(\xi, \eta)}{\partial \eta} d\eta, \\
  dy(\xi, \eta) &= \frac{\partial y}{\partial \xi} d\xi + \frac{\partial y}{\partial \eta} d\eta = \sum_{i} y_i \frac{\partial \phi_i(\xi, \eta)}{\partial \xi} d\xi + \sum_{i} y_i \frac{\partial \phi_i(\xi, \eta)}{\partial \eta} d\eta, \\
  dz(\xi, \eta) &= \frac{\partial z}{\partial \xi} d\xi + \frac{\partial z}{\partial \eta} d\eta = \sum_{i} z_i \frac{\partial \phi_i(\xi, \eta)}{\partial \xi} d\xi + \sum_{i} z_i \frac{\partial \phi_i(\xi, \eta)}{\partial \eta} d\eta.
\end{align*}
\]

(3.57)

Thus a differential face area \( dA \) in the \( (x, y, z) \) space can be written in terms of the reference coordinates \( (\xi, \eta) \),

\[
dA(\xi, \eta) = |J_\Gamma(\xi, \eta)| d\xi d\eta,
\]

(3.58)
with the Jacobian of the transformation \(|J_T(\xi, \eta)|\) calculated by

\[
|J_T(\xi, \eta)| = \left[ \left( \frac{\partial x}{\partial \xi} \right) \left( \frac{\partial x}{\partial \xi} \right)^2 + \left( \frac{\partial x}{\partial \eta} \right) \left( \frac{\partial x}{\partial \eta} \right)^2 + \left( \frac{\partial x}{\partial y} \right) \left( \frac{\partial x}{\partial y} \right)^2 \right]^{1/2},
\]

(3.59)
in which all the partial derivative entries of the matrices can be calculated according to Eq. 3.57.

If it is a linear face, i.e., 3-node triangle or a bilinear face, i.e., 4-node quadrilateral, \(|J_T(\xi, \eta)|\) is nothing but the ratio of the area of face in physical space to the area of face in reference space.

Moreover, the unit vector normal to the differential face area \(dA\) can be calculated by

\[
n_r = \frac{1}{|J_T(\xi, \eta)|} \left( \begin{array}{ccc} \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} & \frac{\partial x}{\partial \xi} \\ \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial y} & \frac{\partial z}{\partial y} & \frac{\partial x}{\partial y} \end{array} \right).
\]

(3.60)

Similarly, \(n_r\) is a constant vector normal to a linear/bilinear face. Note that the grid topology should guarantee that the direction of \(n\) points outward of its host cell.

Shape functions for triangular faces can also be expressed in terms of barycentric coordinates \((L_1, L_2, L_3)\) (or area coordinates):

\[
\phi_i(\xi, \eta) = \phi_i(L_1, L_2, L_3), \quad \text{with} \quad L_1 + L_2 + L_3 = 1.
\]

which are discussed in detail in §B.1.1. Accordingly, the 3-node shape functions and derivatives for linear triangle and 6-node shape functions for curvilinear triangle are expressed in §B.1.2 and §B.1.3, respectively.

If extended to domain calculations, the coordinate variables \(x, y\) and \(z\) are treated as functions of local reference coordinates \(\{(\xi, \eta, \zeta) \mid \xi, \eta, \zeta \in [-1, +1]\}\) on \(\Omega\), then approximated using the standard finite element shape functions in the form

\[
x(\xi, \eta, \zeta) = \sum_{i=1}^{N_e} x_i \phi_i(\xi, \eta, \zeta), \quad y(\xi, \eta, \zeta) = \sum_{i=1}^{N_e} y_i \phi_i(\xi, \eta, \zeta), \quad z(\xi, \eta, \zeta) = \sum_{i=1}^{N_e} z_i \phi_i(\xi, \eta, \zeta),
\]

(3.61)

where \(N_e\) is equal to the number of the shape functions for the element. The 8-node shape functions and derivatives for linear hexahedron and 20-node shape functions and derivatives for trilinear hexahedron are expressed in §B.4.1 and §B.4.2, respectively. The functions \(x, y\) and \(z\)
are differentiable with respect to the coordinates $\xi$, $\eta$ and $\zeta$ in reference space:

\[
\begin{align*}
    dx(\xi, \eta, \zeta) &= \frac{\partial x}{\partial \xi} d\xi + \frac{\partial x}{\partial \eta} d\eta + \frac{\partial x}{\partial \zeta} d\zeta = \sum_{i}^{N_e} x_i \frac{\partial \phi_i}{\partial \xi} d\xi + \sum_{i}^{N_e} x_i \frac{\partial \phi_i}{\partial \eta} d\eta + \sum_{i}^{N_e} x_i \frac{\partial \phi_i}{\partial \zeta} d\zeta, \\
    dy(\xi, \eta, \zeta) &= \frac{\partial y}{\partial \xi} d\xi + \frac{\partial y}{\partial \eta} d\eta + \frac{\partial y}{\partial \zeta} d\zeta = \sum_{i}^{N_e} y_i \frac{\partial \phi_i}{\partial \xi} d\xi + \sum_{i}^{N_e} y_i \frac{\partial \phi_i}{\partial \eta} d\eta + \sum_{i}^{N_e} y_i \frac{\partial \phi_i}{\partial \zeta} d\zeta, \\
    dz(\xi, \eta, \zeta) &= \frac{\partial z}{\partial \xi} d\xi + \frac{\partial z}{\partial \eta} d\eta + \frac{\partial z}{\partial \zeta} d\zeta = \sum_{i}^{N_e} z_i \frac{\partial \phi_i}{\partial \xi} d\xi + \sum_{i}^{N_e} z_i \frac{\partial \phi_i}{\partial \eta} d\eta + \sum_{i}^{N_e} z_i \frac{\partial \phi_i}{\partial \zeta} d\zeta.
\end{align*}
\]

(3.62)

Then a differential domain volume $dV$ in the $(x, y, z)$ space can be written in terms of the reference coordinates $(\xi, \eta, \zeta)$,

\[
dV(\xi, \eta, \zeta) = |J_\Omega(\xi, \eta)||d\xi d\eta d\zeta|,
\]

(3.63)

with the Jacobian of the transformation $|J_\Omega(\xi, \eta)|$ calculated by

\[
|J_\Omega(\xi, \eta, \zeta)| = \begin{vmatrix}
    \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\
    \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\
    \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta}
\end{vmatrix},
\]

(3.64)

in which all the partial derivative entries of the matrices can be calculated according to Eq. 3.62.

Similar to the linear triangular elements, shape functions for tetrahedral elements can also be expressed in terms of barycentric coordinates $(L_1, L_2, L_3, L_4)$:

\[
\phi_i(\xi, \eta, \zeta) = \phi_i(L_1, L_2, L_3, L_4), \quad \text{with} \quad L_1 + L_2 + L_3 + L_4 = 1.
\]

(3.65)

which are discussed in detail in §B.3.1. Accordingly, the 4-node shape functions and derivatives for linear tetrahedron and 10-node shape functions and derivatives for bilinear tetrahedron are expressed in §B.3.2 and §B.3.3, respectively. In addition, the shape functions for pyramid and prism (wedge) are also listed in Appendix B. Some other nonstandard types of element like wrick (prism + pyramid) are not included in our study, since they are rarely used for hybrid grids in CFD.

Numerical integration over a control face, i.e., inviscid flux integration over a triangle, can
be performed with respect to the reference coordinates \((\xi, \eta)\):

\[
\int_{\Gamma_e} \mathbf{F}_k(\mathbf{U}_h) \mathbf{n}_k B_i \ d\Gamma = \int_{\Gamma_e} \mathbf{H}_k \left( \sum_{j=1}^{N} B_j^L \mathbf{u}_j^L, \sum_{j=1}^{N} B_j^R \mathbf{u}_j^R, \mathbf{n}_k \right) B_i^L \ d\Gamma \\
= \sum_{l=1}^{M} \mathbf{H}_k \left( \sum_{j=1}^{N} B_j^L \mathbf{u}_j^L, \sum_{j=1}^{N} B_j^R \mathbf{u}_j^R, \mathbf{n}_k \right) B_i^L \ w_l |J_{\Gamma_e}(\xi_l, \eta_l)|, \quad 1 \leq i \leq N. \tag{3.66}
\]

where the basis functions \(B^L\) and \(B^R\) are the functions of the coordinates \((x, y, z)\), and thus also the functions of the integration points \((\xi, \eta)\) in reference space. In Eq. 3.66, \(w_l\) are the associated weighting factors, and \(M\) is the total number of integration points, which indicates the integration order as listed in Table C.1. For example, if \(M = 1\), an exact integration can be obtained for a polynomial of degree \(p = 1\); for \(M = 3\), integration is exact for \(p \leq 2\); and for \(M = 4\), \(p \leq 3\). The weights also exist for quadrilateral faces, which are listed in Table C.2.

Similarly, numerical integration over a control volume, i.e., inviscid flux integration over a tetrahedron can be conducted using the reference coordinates \((\xi, \eta, \zeta)\):

\[
\int_{\Omega_e} \mathbf{F}_k(\mathbf{U}_h) \frac{\partial B_i}{\partial x_k} \ d\Omega = \sum_{l=1}^{M} \mathbf{F}_k \left( \sum_{j=1}^{N} B_j \mathbf{u}_j \right) \frac{\partial B_i}{\partial x_k} \ w_l |J_{\Omega_e}(\xi_l, \eta_l, \zeta_l)|, \quad 1 \leq i \leq N. \tag{3.67}
\]

The procedures for volume integration are basically the same for tetrahedron, pyramid, prism and hexahedron, and the weighting coefficients for these types of elements can be found in Appendix C.

### 3.5 Boundary Conditions

All the boundary conditions are weakly imposed in this work. Different from using the so-called ghost boundary cell in the finite volume methods, the DG methods usually employ the so-called boundary state vector \(\mathbf{U}_b^h(\mathbf{U}_h^-, \mathbf{U}_\infty)\) constructed at the boundary integration point, which is a function of the state vector \(\mathbf{U}_h^-\) of interior cell and the known free-stream vector \(\mathbf{U}_\infty\).

#### 3.5.1 Characteristic Boundary

If a boundary face is tagged as “inflow” or “outflow” or “farfield” in a grid, it will be uniformly treated as a characteristic boundary in the solver. The actual flow condition at the boundary face is determined by the Mach number \(M^b\) derived from the mean state vector \((P_0)\) in the
interior cell. Four conditions are categorized based on the value of $M^b$, and then the $\mathbf{U}^b_h$ vector and the approach of flux evaluation are specified accordingly.

**supersonic inflow ($M^b \leq -1$)**

$$\mathbf{H}^b = \mathbf{F}(\mathbf{U}_\infty), \quad (3.68)$$

where the first-order free-stream values $\mathbf{U}_\infty$ are prescribed at the integration point of the boundary face for flux evaluation.

**subsonic inflow ($-1 < M^b < 0$)**

$$\mathbf{U}^b_h = \begin{pmatrix} \rho^b \\ \rho u^b \\ \rho v^b \\ \rho w^b \\ \rho e^b \end{pmatrix}, \quad \mathbf{H}^b = \mathbf{H}(\mathbf{U}^b_h, \mathbf{U}^b_h, \mathbf{n}^b), \quad (3.69)$$

where $\mathbf{U}_\infty$ are prescribed to the boundary state vector at the integration point, and the boundary flux is evaluated using a numerical flux scheme.

**subsonic outflow ($0 \leq M^b < 1$)**

$$\mathbf{U}^b_h = \begin{pmatrix} \rho^- \\ \rho u^- \\ \rho v^- \\ \rho w^- \\ \gamma (p^- - \frac{1}{2} \rho^- |\mathbf{V}^-|^2) \end{pmatrix}, \quad \mathbf{H}^b = \mathbf{F}(\mathbf{U}^b_h), \quad (3.70)$$

where $\mathbf{V}^- = (u^-, v^-, w^-)$, and the first-order free-stream pressure $p^-\infty$ is imposed for the boundary state vector at the integration point, and the rest of required variables are computed from the interior cell.

**supersonic outflow ($M^b \geq 1$)**

$$\mathbf{H}^b = \mathbf{F}(\mathbf{U}_h^-), \quad (3.71)$$

where the interior state vector is extrapolated at the integration point of the boundary face and used for evaluation of the boundary flux.
3.5.2 Slip Wall / Symmetry Boundary

At the slip wall / symmetry boundary, the flow tangency requirement \( \mathbf{V} \cdot \mathbf{n} = 0 \) needs to be satisfied at the integration point. Instead of using the face unit normal vector \( \mathbf{n}^b \), the approximate physical unit normal vector \( \mathbf{n}^g \), which is a more accurate representation of normal direction on the physical entity, is required for the solver to obtain steady convergence and formal order of accuracy [68]. The vector \( \mathbf{n}^g \) is obtained through a quadratic representation of the solid wall boundary, i.e., 6-node curvilinear triangle and 8-node curvilinear quadrilateral as shown in Fig. B.4 and Fig. B.6, which is adopted in accordance with the quadratic spatial discretization. A comprehensive description of various types of higher-order elements can be found in Appendix B. Unlike those approaches that remodel and approximate the curved geometries with either additional geometric information or extended stencils on the linear elements [68,69], i.e., 3-node triangle and 4-node bilinear quadrilateral, in this work the curved triangular or quadrilateral surface meshes are directly generated provided with the CAD geometric configurations during pre-processing, indicating the robustness of the code to deal with arbitrary complex geometries without recourse to any special treatment. Thus \( \mathbf{n}^g \) is approximated by Eq. 3.60 at each integration point.

The velocity at the integration point is then computed as

\[
\begin{align*}
\mathbf{u}^b &= \mathbf{u}^- - 2 (\mathbf{V}^- \cdot \mathbf{n}^g) n_x^g, \\
\mathbf{v}^b &= \mathbf{v}^- - 2 (\mathbf{V}^- \cdot \mathbf{n}^g) n_y^g, \\
\mathbf{w}^b &= \mathbf{w}^- - 2 (\mathbf{V}^- \cdot \mathbf{n}^g) n_z^g.
\end{align*}
\]

The density and total energy at the integration point are computed from the interior cell. The boundary state vector and the numerical flux is then computed as

\[
\mathbf{U}^b_h = \begin{pmatrix}
\rho^- \\
\rho^- u^b \\
\rho^- v^b \\
\rho^- w^b \\
\rho^- e^-
\end{pmatrix}, \quad \mathbf{H}^b = \mathbf{H}(\mathbf{U}_h^-, \mathbf{U}_h^b, \mathbf{n}^b),
\]

where one can see that \( \mathbf{n}^g \) is only required to compute \( \mathbf{V}^b \), and \( \mathbf{n}^b \) is still used in flux evaluation.
3.5.3 No-Slip Adiabatic Wall Boundary

At the no-slip adiabatic wall boundary, the zero velocity condition and the zero normal temperature gradient condition at the wall face are prescribed at the integration point

\[ u^b = v^b = w^b = 0, \quad \left( \frac{\partial T}{\partial n^b} \right) = (\nabla T)^b \cdot n^b = 0, \] (3.76)

where the rest of the required state variables are computed from the interior cell

\[ U_h^b = \begin{pmatrix} \rho^- \\ 0 \\ 0 \\ 0 \end{pmatrix}. \] (3.77)

To take the possible jump at the wall boundary into consideration, the corrected gradient \((\nabla U_h + r_h)^b\) is adopted as the auxiliary gradient, where \(r_h\) is the so-called local lifting operator computed at the wall boundary [15]. The numerical flux at the no-slip adiabatic wall boundary is computed as

\[ H^b = F \left( U_h^b, (\nabla U_h + r_h)^b \right). \] (3.78)

3.5.4 No-Slip Isothermal Wall Boundary

At the no-slip isothermal wall boundary, the zero velocity condition and the fixed temperature \(T_{wall}\) condition at the wall face are prescribed at the integration point

\[ u^b = v^b = w^b = 0, \quad T^b = 2T_{wall} - T^- \quad \rightarrow \rho e^b = \frac{\rho^- \bar{R} T^b}{\gamma - 1}, \] (3.79)

where \(\bar{R}\) is the non-dimensional universal gas constant for perfect gas, and the rest of the required state variables are computed from the interior cell

\[ U_h^b = \begin{pmatrix} \rho^- \\ 0 \\ 0 \\ \rho e^b \end{pmatrix}. \] (3.80)
Similar to the adiabatic wall treatment, the corrected gradient \((\nabla U_h + r_h)^b\) is used to account for the possible jump at the wall boundary. The numerical flux at the no-slip isothermal wall boundary is computed as

\[
H^b = F\left( U^b_h, (\nabla U_h + r_h)^b \right).
\] (3.81)

### 3.5.5 Periodic Boundary

In practical applications [62], the applicability of periodic boundary condition is restricted to flow configurations that are indeed periodic owing to their geometry, such as flow around turbine blades, or flow through channel, pipe, and duct with one or more statistically homogeneous flow directions. For time-dependent DNS or LES, a periodic boundary is often encountered, e.g., “Taylor-Green Vortex”, and “turbulent channel flow”. A periodic boundary may be regarded as an overlapping inflow and outflow boundary or an internal face boundary. Indeed, an internal face boundary does not bring in any challenge, since that the flow field through the ghost cell of one periodic boundary outside the domain is known from the computations on the interior cell adjacent to the opposing periodic boundary inside the domain. Therefore, the specification of periodic boundaries should be quite straightforward. However, care should be taken when it comes to parallel computation. In the parallel mode, the most important step is about how to identify the opposing periodic boundary face of the given face, since the periodic boundary face and its opposing one may not locate in the same partition. The crucial clue is that the global numbering of the boundary is fixed in spite of the number of processors.

In our work, with MPI routines, the periodic boundary condition are applied as follows. First, the global numbering of the boundary face will be gathered from every processor to the root processor using the collective communication routine \textit{MPI gatherv}. This mapping information would be stored in a mapping array. Next, this mapping array will be sent to each processor from the root processor using the collective communication routine \textit{MPI bcast}. Once we have this array in each processor, the opposing periodic boundary face can be determined by simple conditional expressions in the loop. The numerical flux at the periodic boundary is computed the same way as the internal face with the information specified from the found opposing periodic boundary face.
Chapter 4

Hyperbolic Reconstructed Discontinuous Galerkin Methods

The first order hyperbolic system (FOHS) was first introduced by Nishikawa as a radical approach for solving diffusion problem [109]. By including derivative quantities as additional variables, the equations are first formulated as a first order system (FOS). Then, it is rendered to be hyperbolic, which is the distinguished feature of the FOHS method from other FOS methods, by adding pseudo time derivatives to the first-order system. It thus generates a system of pseudo-time evolution equations for the solution and the derivatives in the partial differential equation (PDE) level, not in the discretization level as in DG methods. Due to the fact that the well-established methods can be directly applied to the viscous terms in the FOHS, the formulation in the PDE level would allow a dramatic simplification in the discretization. The FOS method is especially attractive in the context of the DG methods, since it allows the use of inviscid algorithms for the viscous terms and thus greatly simplifies the discretization of the compressible Navier-Stokes equations. 4

4.1 First Order Hyperbolic System Formulation

Consider the following model linear advection-diffusion equation in 2D.

\[
\frac{\partial \varphi}{\partial t} + a \frac{\partial \varphi}{\partial x} + b \frac{\partial \varphi}{\partial y} = \nu \left( \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} \right) + f(x, y),
\]

(4.1)
where \( \varphi \) denotes a scalar function that can be referred to as the primary solution variable (or the velocity potential), \((a, b)\) is a constant advection vector, \( \nu \) is a positive diffusion coefficient, and \( f(x, y) \) is the source term. In order to reformulate this equation into a first-order hyperbolic advection diffusion system, derivatives of the primary solution variable \( \varphi \) would be needed as additional variables. Therefore, the gradient variables (or the velocity vector) \( \mathbf{v} \) is defined as

\[
\mathbf{v} = \nabla \varphi = \begin{pmatrix} v_x \\ v_y \end{pmatrix}, \tag{4.2}
\]

where \( v_x \) and \( v_y \) are the components of the gradient variables.

By adding pseudo time derivatives with respect to all variables, the following first-order hyperbolic system for this advection-diffusion equation can be formulated.

\[
\begin{aligned}
\frac{\partial \varphi}{\partial \tau} + a \frac{\partial \varphi}{\partial t} + b \frac{\partial \varphi}{\partial y} + \nu \left( \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} \right) &= f(x, y), \\
\frac{\partial v_x}{\partial \tau} &= \frac{1}{T_r} \left( \frac{\partial \varphi}{\partial x} - v_x \right), \\
\frac{\partial v_y}{\partial \tau} &= \frac{1}{T_r} \left( \frac{\partial \varphi}{\partial y} - v_y \right),
\end{aligned}
\tag{4.3}
\]

where \( t \) and \( \tau \) are understood as the physical time and the pseudo time respectively. Clearly, the gradient variables would relax to the solution derivatives in the steady state of pseudo time, leading to the consistent gradients at any instant of physical time. Here, \( T_r \) is a free parameter, named as relaxation time. Note that the system is equivalent to the original advection-diffusion equation in the steady state for any nonzero \( T_r \), but \( T_r \) needs to be positive for the system to be hyperbolic. For steady problems, the system without the physical time derivative can be solved by marching in the pseudo time to yield a steady solution to the original equation. As for unsteady problems, the physical time derivative is discretized by the ESDIRK (Explicit first stage, Single Diagonal coefficient, diagonally Implicit Runge-Kutta) method, while the pseudo time may be discretized by BDF1 to march in \( \tau \) towards a pseudo steady state. The details would be discussed later.

At this point, it is noticed that the FOHS formulation has introduced two extra variables, \( v_x \) and \( v_y \), and a numerical scheme would involve two additional equations compared with a scheme applied to the original scalar equation, i.e., Eq. 4.1. Seemingly, the FOHS formulation requires more computational efforts than a conventional scheme. In FV methods, the number of discrete equations would increase by the number of derivatives for each variable, e.g., fifteen extra equations in the case of the three-dimensional Navier-Stokes equations \([107, 117]\). Note that
this is equivalent to P₁ DG methods, which introduce fifteen derivatives in the discretization level: both Hyperbolic-Navier-Stokes (HNS) FV and P₁ DG methods require twenty discrete unknowns. However, the HNS FV method is not necessarily more expensive than a conventional FV method because the resulting schemes achieve $O(1/h)$ speed-up in iterative convergence by the elimination of a typical $O(1/h^2)$ diffusion stiffness, and also yield one-order higher order accuracy in the solution gradients and in the advective/inviscid approximation as demonstrated in Refs. [78,114,115,116,117]. In DG methods, these extra variables and their moments can be used to build/replace the high-order moments in the primary solution polynomial; thus the order of polynomial of the primary solution variable is always $P_0$. As a result, it leads to a scheme achieving a comparable level of accuracy as a conventional DG scheme for the same number of degrees of freedom as mentioned in Introduction. Nevertheless, the resulting scheme has one-order-lower order of accuracy in the diffusion term, and most importantly does not contribute to reducing the cost of the DG methods [104]. The objective of the present study is to demonstrate that the FOHS method combined with the rDG method can break this barrier and generate schemes truly more efficient than conventional DG methods.

The FOHS can be written in the vector form as

$$\frac{\partial \mathbf{U}}{\partial \tau} + \mathbf{T} \frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}_x}{\partial x} + \frac{\partial \mathbf{F}_y}{\partial y} = \mathbf{S},$$  \hspace{1cm} (4.4)

where

$$\mathbf{U} = \begin{pmatrix} \varphi \\ v_x \\ v_y \end{pmatrix}, \quad \mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$\mathbf{F}_x = \begin{pmatrix} a \varphi - \nu v_x \\ -\varphi/T_r \\ 0 \end{pmatrix}, \quad \mathbf{F}_y = \begin{pmatrix} b \varphi - \nu v_y \\ 0 \\ -\nu v_x/T_r \end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix} f(x,y) \\ -v_x/T_r \\ -v_y/T_r \end{pmatrix}.$$ \hspace{1cm} (4.6)

In this paper, we consider the advection term and the diffusive term separately.

$$\mathbf{F}_x = \mathbf{F}_x^a + \mathbf{F}_x^d = \begin{pmatrix} a \varphi \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -\nu v_x \\ -\varphi/T_r \\ 0 \end{pmatrix}, \quad \mathbf{F}_y = \mathbf{F}_y^a + \mathbf{F}_y^d = \begin{pmatrix} b \varphi \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -\nu v_y \\ 0 \\ -\varphi/T_r \end{pmatrix}.$$ \hspace{1cm} (4.7)

Consider the Jacobian of the flux projected along $\mathbf{n} = (n_x, n_y)$,

$$\mathbf{A}_n = \frac{\partial \mathbf{F}_x}{\partial \mathbf{U}} n_x + \frac{\partial \mathbf{F}_y}{\partial \mathbf{U}} n_y = \mathbf{A}_n^a + \mathbf{A}_n^d,$$ \hspace{1cm} (4.8)
where $A^a_n$ and $A^d_n$ are the advective and diffusive Jacobians, respectively.

$$A^a_n = \frac{\partial F^a_x}{\partial U} n_x + \frac{\partial F^a_y}{\partial U} n_y = \begin{pmatrix} a_n & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad (4.9)$$

$$A^d_n = \frac{\partial F^d_x}{\partial U} n_x + \frac{\partial F^d_y}{\partial U} n_y = \begin{pmatrix} 0 & -\nu n_x & -\nu n_y \\ -n_x/T_r & 0 & 0 \\ -n_y/T_r & 0 & 0 \end{pmatrix}, \quad (4.10)$$

and

$$a_n = an_x + bn_y. \quad (4.11)$$

The only non-zero eigenvalue of advective Jacobian is $a_n$, while the diffusive Jacobian has the following eigenvalues

$$\lambda_1 = \sqrt{\frac{\nu}{T_r}}, \quad \lambda_2 = -\sqrt{\frac{\nu}{T_r}}, \quad \lambda_3 = 0. \quad (4.12)$$

The first two nonzero eigenvalues indicate that the system describes a wave propagating isotropically if we only consider the diffusive part. The third eigenvalue corresponds to the inconsistency damping mode [109]. The relaxation time $T_r$ does not affect the steady solution, and thus can be defined solely for the purpose of accelerating the convergence to the steady state. For simplicity, $T_r$ is defined as

$$T_r = \frac{L_r^2}{\nu}, \quad L_r = \frac{1}{\max(Re, 2\pi)}, \quad Re = \frac{\sqrt{a^2 + b^2}}{\nu}. \quad (4.13)$$

Note that we include Reynolds (or Péclet) number information in the relaxation length scale, so that the developed method could deliver the designed order of accuracy with fast convergence when it comes to narrow boundary layer type problem. See Ref. [118] for details.

This study can be easily extend to other model equations. For instance, one can consider diffusion equations with a tensor coefficient, i.e.,

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (D \nabla \phi) + f(x, y), \quad (4.14)$$

where $D$ denotes a symmetric positive definite diffusion tensor, which can be expressed as

$$D = \begin{pmatrix} D_{xx}(x, y) & D_{xy}(x, y) \\ D_{yx}(x, y) & D_{yy}(x, y) \end{pmatrix}. \quad (4.15)$$
Thus, a similar procedure can be used to derive the FOHS. Unlike the previous advection-diffusion case, the advection part of the flux would change to zero while the diffusive flux is given as

\[
\begin{pmatrix}
-D_{xx}v_x - D_{xy}v_y \\
-\varphi/T_r \\
0
\end{pmatrix}, \quad \begin{pmatrix}
-D_{xy}v_x - D_{yy}v_y \\
0 \\
-\varphi/T_r
\end{pmatrix},
\]

Since \(D\) is a symmetric positive definite tensor, consider an unit vector \(n\) at an arbitrary direction, one would have

\[
\nu = n^T D n = D_{xx}n_x^2 + 2D_{xy}n_xn_y + D_{yy}n_y^2 > 0.
\]

Therefore, the Jacobian of the flux projected along \(n\),

\[
A_n = \frac{\partial(F \cdot n)}{\partial U} = \begin{pmatrix}
0 & -D_{xx}n_x - D_{xy}n_y & -D_{xy}n_x - D_{yy}n_y \\
-n_x/T_r & 0 & 0 \\
-n_y/T_r & 0 & 0
\end{pmatrix}.
\]

It has the following eigenvalues

\[
\lambda_1 = -\sqrt{\frac{\nu}{T_r}}, \quad \lambda_2 = -\sqrt{\frac{\nu}{T_r}}, \quad \lambda_3 = 0.
\]

Clearly, the eigen-structure of this system is very similar to \(A_n^d\) we just derived for the advection-diffusion equation.

The above-mentioned two cases is linear equations only. Extensions to nonlinear diffusion equations require additional considerations in the construction of numerical schemes. It is necessary to consider the nonlinear diffusion equation since the diffusive flux in the Navier-Stokes equation is also nonlinear.

Therefore, let us consider the following nonlinear diffusion equations.

\[
\frac{\partial \varphi}{\partial t} = \frac{\partial}{\partial x} \left( \nu(\varphi) \frac{\partial \varphi}{\partial x} \right) + \frac{\partial}{\partial y} \left( \nu(\varphi) \frac{\partial \varphi}{\partial y} \right)
\]

where \(\nu\) is a positive diffusion coefficient depending on the solution \(\varphi\).

Following the similar procedure, one should arrived the corresponding first order hyperbolic
system. Consider the Jacobian of the flux projected along $\mathbf{n} = (n_x, n_y)$,

$$A_n = \frac{\partial \mathbf{F}}{\partial \mathbf{U}} n_x + \frac{\partial \mathbf{F}}{\partial \mathbf{U}} n_y + \frac{\partial \mathbf{F}}{\partial \mathbf{U}} n_z = A^L_n + A^N_n,$$

(4.21)

where $A^L_n$ and $A^N_n$ are the linear part and the nonlinear part of Jacobians, respectively.

$$A^L_n = \begin{pmatrix} 0 & -\nu n_x & -\nu n_y \\ -n_x/T_x & 0 & 0 \\ -n_y/T_y & 0 & 0 \end{pmatrix}, \quad A^N_n = \begin{pmatrix} -\frac{\partial \nu}{\partial \varphi}(\mathbf{v}_k \mathbf{n}_k) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

(4.22)

### 4.2 Weak Formulation

The FOHS of equations, i.e. Eq. (4.4), can be discretized using a discontinuous Galerkin finite element formulation. We assume that the domain $\Omega$ is subdivided into a collection of non-overlapping arbitrary elements (or cells) $\{\Omega_i\}$, and then introduce the following broken Sobolev space $V^n_h$

$$V^n_h = \left\{ v_h \in [L^2(\Omega)]^k : v_h|_{\Omega_i} \in \left[ V^n_k \right] \forall \Omega_i \in \Omega \right\},$$

(4.23)

which consists of discontinuous vector polynomial functions of degree $n$, and where $k$ is the dimension of the unknown vector and $V^n$ is the space of all polynomials of degree $\leq n$. To formulate the discontinuous Galerkin method, we introduce the following weak formulation, which is obtained by multiplying Eq. (4.4) by a test function $W_h$, integrating over an element $\Omega_i$, and then performing an integration by parts: find $U_h \in V^n_p$ such as

$$\frac{\partial}{\partial \tau} \int_{\Omega_i} W_h U_h d\Omega + \frac{\partial}{\partial t} \int_{\Omega_i} W_h T U_h d\Omega$$

$$+ \int_{\Gamma_i} W_h \mathbf{F}_k n_k d\Gamma - \int_{\Omega_i} \frac{\partial W_h}{\partial x_k} \mathbf{F}_k d\Omega = \int_{\Omega_i} W_h S d\Omega, \quad \forall W_h \in V^n_h,$$

(4.24)

where $U_h$ and $W_h$ are represented by piecewise polynomial functions of degrees $p$, which are discontinuous between the cell interfaces, and $\mathbf{n}_k$ the unit outward normal vector to the $\Gamma_i$: the boundary of $\Omega_i$. The standard DG solution $U_h$ within the element $\Omega_i$ can be expressed as

$$U_h(x, y, t, \tau) = C(x, y) V(t, \tau),$$

(4.25)

where $C$ is a basis matrix, and $V$ is a vector of unknown polynomial coefficients. A further discussion about $C$ and $V$ will be given later in this section.
If we set the test function $W_h$ as the transpose of the basis matrix $C$, then the following equivalent system would be arrived.

$$
\frac{\partial}{\partial \tau} \int_{\Omega_i} C^T C V d\Omega + \frac{\partial}{\partial t} \int_{\Omega_i} C^T T C V d\Omega + \int_{\Gamma_i} C^T F_K^n d\Gamma - \int_{\Omega_i} \frac{\partial C^T}{\partial x_k} F_K d\Omega = \int_{\Omega_i} C^T S d\Omega. \quad (4.26)
$$

Since the numerical solution $U_h$ is discontinuous between element interfaces, the interface fluxes are not uniquely defined. This scheme is called the discontinuous Galerkin method of degree $n$, or in short notation DG($P_n$) method. By simply increasing the degree $n$ of the polynomials, the DG methods of corresponding higher order are obtained.

Compared with reconstructed FV methods, the DG methods would require more degrees of freedom, additional domain integration, and more Gauss quadrature points for the boundary integration, which leads to more computational costs and storage requirements. Inspired by the reconstructed DG methods from Dumbser et al. in the frame of $P_n P_m$ scheme [39, 40, 41], termed rDG($P_n P_m$) in this paper, least-squares based and variational reconstruction based rDG methods are designed to achieve high order of accuracy while reducing the computational cost. In fact, a unified formulation would be provided by rDG method for both FV and DG methods. The standard FV and DG methods would be nothing but special cases in rDG framework, thus allowing for a direct efficiency comparison. For rDG($P_n P_m$) method with $m > n$, a higher-order reconstructed numerical solution is constructed over an element $\Omega_i$:

$$
U_R^h(x, y, t, \tau) = C^R(x, y)V^R(t, \tau), \quad (4.27)
$$

where the superscript $R$ indicates reconstructed polynomials, and higher-order derivatives (higher than $n$-th and up to $m$-th) are reconstructed from the underlying $P_n$ polynomial. This higher-order numerical solution $U_R^h$ is used for flux and source term computations in order to raise the order of accuracy. There are three approaches to the reconstruction. One is a least-squares reconstruction method, and another is a variational reconstruction method. The variational reconstruction generates a globally coupled system of equations for gradients by minimizing jumps in the solution and derivatives at element interfaces [147]. The resulting linear system is iteratively solved along with the solution iteration, and therefore the cost is comparable to a least-squares reconstruction. See Ref. [147] for details. The method based on the least-squares/variational reconstruction is expressed by rDG($P_n P_m$). The third approach, which is unique in the FOHS formulation considered here, is to directly use the gradient variables and their moments to evaluate the higher-order derivatives in the primary solution polynomial. Or equivalently, this approach can be thought of as defining the primary solution as $P_m$, and use
the higher-order moments to represent the gradient variables in the FOHS formulation. This is the key idea to effectively reducing the number of discrete unknowns despite the increase in the variables in the FOHS formulation. The method based on this approach is expressed by DG($P_0P_m$). Naturally, rDG($P_nP_m$) and DG($P_0P_m$) can be combined to generate efficient schemes as we will discuss later.

By moving the third and fourth terms to the right-hand-side (r.h.s.) in Eq. (4.26), we will arrive at

$$M_r \frac{\partial V}{\partial \tau} + M_t \frac{\partial V}{\partial t} = R(U_h^R), \quad (4.28)$$

where $M_r$ and $M_t$ are the mass matrices defined as,

$$M_r = \int_{\Omega_i} C^T C d\Omega, \quad (4.29)$$

$$M_t = \int_{\Omega_i} C^T T C d\Omega, \quad (4.30)$$

and $R$ is the residual vector, defined as

$$R = \int_{\Omega_i} \frac{\partial C^T}{\partial x_k} F_k(U_h^R) + C^T S(U_h^R) d\Omega - \int_{\Gamma_i} C^T F_k(U_h^R) n_k d\Gamma. \quad (4.31)$$

### 4.3 Spatial Discretization

Based on different rDG methods, some effective discretization hyperbolic rDG methods will be presented to deal with the derived FOHS. The format $A + B$ is used to indicate the discretization method for the system, where $A$ refers to the discretization method for $\varphi$ and $B$ refers to the discretization method for its derivatives. Note that what make the hyperbolic DG methods based on FOHS different from standard DGMs is that the system is in the partial differential equation (PDE) level other than in the discretization level. While the choice of $B$ would be either standard DG($P_n$) method or rDG($P_nP_m$) method, we have one more option in $A$. Since FOHS would introduce the derivatives into the system as auxiliary variables, one can construct higher order polynomial in the primary variable while keeping the minimal degrees of freedom. In FV methods, this technique corresponds to Scheme-II in Ref. [114], which replaces the LS gradients in the primary variable reconstruction by the auxiliary variables. It thus eliminates the need for the LS gradient computation for the primary variable, but the degrees of freedom remains the same. In the framework of the DG methods, it eliminates the need to store and
solve for high-order moments for the primary variable \([104]\), and effectively reducing the degrees of freedom and the number of discrete equations. This new approach, termed \(\text{DG}(P_0 P_k)\), can deliver high order accuracy in \(\varphi\) with the minimal computational cost and storage requirement, where \(k\) is the order of polynomial we construct for \(\varphi\) with the information from its derivatives.

The following combinations of \(\mathbf{A} + \mathbf{B}\) are listed to illustrate the different hyperbolic rDG methods we discuss in this work.

- \(\text{DG}(P_n) + \text{DG}(P_n)\) and \(\text{rDG}(P_n P_m) + \text{rDG}(P_n P_m)\)

These combinations would use standard \(\text{DG}(P_n)\) and \(\text{rDG}(P_n P_m)\) methods, respectively, for both primary variable \(\varphi\) and its first derivatives.

- \(\text{DG}(P_0 P_{n+1}) + \text{DG}(P_n)\) and \(\text{rDG}(P_0 P_{m+1}) + \text{rDG}(P_n P_m)\)

Note that with the information of the derivatives in hand, one can easily construct higher order of polynomial for \(\varphi\) without increasing the degrees of freedom for \(\varphi\) nor solving an over-determined system to reconstruct high-order moments. Therefore, these combinations would use standard \(\text{DG}(P_n)/\text{rDG}(P_n P_m)\) methods for the derivatives, while the \(\varphi\) is constructed to one order higher than its derivatives. To minimize the memory and storage cost of the developed methods, one can apply \(\text{DG}(P_n)\) or \(\text{rDG}(P_n P_m)\) methods only on the gradient variables. On the other hand, a higher order of polynomial for \(\varphi\) can be constructed with only one degree of freedom. Therefore, in this paper, we focus on \(\text{DG}(P_0 P_{n+1}) + \text{DG}(P_n)\) and \(\text{DG}(P_0 P_{m+1}) + \text{rDG}(P_n P_m)\) methods.

In the implementation of the DG methods in this work, modal-basis DG methods are adopted. The numerical polynomial solutions are represented using a Taylor series expansion at the cell center and normalized to improve the conditioning of the system matrix: e.g., in two dimensions,

\[
\varphi_h = \overline{\varphi} + \varphi^c_x \Delta x B_2 + \varphi^c_y \Delta y B_3 + \varphi^c_{xx} \Delta x^2 B_4 + \varphi^c_{yy} \Delta y^2 B_5 + \varphi^c_{xy} \Delta x \Delta y B_6 \\
+ \varphi^c_{xxx} \Delta x^3 B_7 + \varphi^c_{yyy} \Delta y^3 B_8 + \varphi^c_{xxy} \Delta x^2 \Delta y B_9 + \varphi^c_{xyy} \Delta x \Delta y^2 B_{10} + \cdots ,
\]

(4.32)

where the \(\overline{\varphi}\) represents the cell-averaged quantity of \(\varphi\), and the superscript \(c\) stands for the
central values. The basis functions are given as follows

\[ B_1 = 1, B_2 = \frac{x - x_c}{\Delta x}, B_3 = \frac{y - y_c}{\Delta y}, B_4 = \frac{1}{2} \left( B_2^2 - \frac{1}{\Omega_i} \int_{\Omega_i} B_2^2 d\Omega \right), \]

\[ B_5 = \frac{1}{2} \left( B_3^2 - \frac{1}{\Omega_i} \int_{\Omega_i} B_3^2 d\Omega \right), B_6 = B_2 B_3 - \frac{1}{\Omega_i} \int_{\Omega_i} B_2 B_3 d\Omega, \]

\[ B_7 = \frac{1}{6} \left( B_2^3 - \frac{1}{\Omega_i} \int_{\Omega_i} B_2^3 d\Omega \right), B_8 = \frac{1}{6} \left( B_3^3 - \frac{1}{\Omega_i} \int_{\Omega_i} B_3^3 d\Omega \right), \]

\[ B_9 = \frac{1}{2} \left( B_2^2 B_3 - \frac{1}{\Omega_i} \int_{\Omega_i} B_2^2 B_3 d\Omega \right), B_{10} = \frac{1}{2} \left( B_2 B_3^2 - \frac{1}{\Omega_i} \int_{\Omega_i} B_2 B_3^2 d\Omega \right). \]  

(4.33)

Here, we have

\[ \Delta x = 0.5(x_{\text{max}} - x_{\text{min}}), \quad \Delta y = 0.5(y_{\text{max}} - y_{\text{min}}), \]

where \( x_{\text{max}}, x_{\text{min}}, y_{\text{max}}, \) and \( y_{\text{min}} \) are used to represent the maximum and minimum coordinates values of the vertexes of the cell.

As we mentioned in the previous work \[81\], one can choose the unknown vector \( \mathbf{V} \) to make the resultant scheme have the same number of degrees of freedom as conventional DG methods for a comparable level of accuracy. As a matter of fact, if a Petrov-Galerkin formulation with a simplified basis function matrix is implemented, one would end up with the same hyperbolic rDG methods we used for diffusion equation \[81\]. On the other hand, if a consistent Galerkin formulation is used, one can make all variables coupled and thus to have better stability properties for advection-diffusion problems. This is the approach taken in this study. Below, some examples of the unknown vector \( \mathbf{V} \) and the basis matrix \( \mathbf{C} \) under Galerkin formulation are shown for better illustration. The first three examples are the hyperbolic DG schemes, and the last two are the hyperbolic rDG schemes, which are more efficient.

- **DG(P₀P₁)+DG(P₀)**

\[
\mathbf{V} = \begin{pmatrix}
\varphi \\
\varphi_x \Delta x \\
\varphi_y \Delta y
\end{pmatrix},
\]

\[
\mathbf{C} = \begin{pmatrix}
B_1 & B_2 & B_3 \\
0 & B_1 \Delta x^{-1} & 0 \\
0 & 0 & B_1 \Delta y^{-1}
\end{pmatrix}.
\]  

(4.35)  

(4.36)
As we can see here, the basis matrix $C$ has included the connection between $\varphi$ and its derivatives, leading to a coupled system. Note that the derivatives of $\varphi$ are expressed by the gradient variables in the FOHS, whereas conventional $P_1$ DG methods determine them as solutions to discrete equations derived by the weak formulation. This particular construction corresponds to a less-efficient variant in Ref. [104]: it has the same number of degrees of freedom as a conventional second-order $P_1$ DG method, but is only first-order accurate for diffusion.

- **DG($P_0P_2$)+DG($P_1$)**

$$V = [\varphi, \overline{\varphi}_x \Delta x, \overline{\varphi}_y \Delta y, \varphi_{xx} \Delta x^2, \varphi_{yy} \Delta y^2, \varphi_{xy} \Delta x \Delta y]^T,$$

$$C = \begin{pmatrix}
B_1 & B_2 & B_3 & B_4 & B_5 & B_6 \\
0 & B_1 \Delta x^{-1} & 0 & B_2 \Delta x^{-1} & 0 & B_3 \Delta x^{-1} \\
0 & 0 & B_1 \Delta y^{-1} & 0 & B_3 \Delta y^{-1} & B_2 \Delta y^{-1}
\end{pmatrix}. \quad (4.38)$$

Compared with authors’ previous work [81], the degrees of freedom for DG($P_0P_2$)+DG($P_1$) has been reduced from 7 to 6 by replacing the redundant cross term with a unified unknown. Hence, this method has the same number of degrees of freedom as $P_2$ conventional DG methods. This scheme, again, corresponds to a less-efficient variant in Ref. [104]: it has the same number of degrees of freedom as a conventional third-order $P_2$ DG method, but is only second-order accurate for diffusion.

- **DG($P_0P_3$)+DG($P_2$)**

$$V = [\varphi, \overline{\varphi}_x \Delta x, \overline{\varphi}_y \Delta y, \varphi_{xx} \Delta x^2, \varphi_{yy} \Delta y^2, \varphi_{xy} \Delta x \Delta y, \varphi_{xxx} \Delta x^3, \varphi_{yyy} \Delta y^3, \varphi_{xxy} \Delta x^2 \Delta y, \varphi_{xyy} \Delta x \Delta y^2]^T,$$

$$C = \begin{pmatrix}
B_1 & 0 & 0 \\
B_2 & B_1 \Delta x^{-1} & 0 \\
B_3 & 0 & B_1 \Delta y^{-1} \\
B_4 & B_2 \Delta x^{-1} & 0 \\
B_5 & 0 & B_3 \Delta y^{-1} \\
B_6 & B_3 \Delta x^{-1} & B_2 \Delta y^{-1} \\
B_7 + B_2 B_4^c & B_4 \Delta x^{-1} & 0 \\
B_8 + B_3 B_5^c & 0 & B_5 \Delta y^{-1} \\
B_9 + B_2 B_6^c + B_3 B_4^c & B_6 \Delta x^{-1} & B_4 \Delta y^{-1} \\
B_{10} + B_2 B_6^c + B_3 B_5^c & B_5 \Delta x^{-1} & B_6 \Delta y^{-1}
\end{pmatrix}. \quad (4.40)$$

63
The complexity in the basis matrix is due to the fact that the average values and cell centers values are not equal. This can be derived using Taylor expansion. One can find a similar procedure in Ref. [104]. Note that it has the same number of degrees of freedom as a conventional fourth-order $P_3$ DG method, but is only third-order accurate for diffusion.

- **DG($P_0P_3$)+rDG($P_0P_1$) - Hyperbolic rDG**

\[
\mathbf{V} = \begin{pmatrix}
\varphi \\
\varphi_x \Delta x \\
\varphi_y \Delta y
\end{pmatrix},
\]

\[
\mathbf{C} = \begin{pmatrix}
B_1 & B_2 & B_3 \\
0 & B_1 \Delta x^{-1} & 0 \\
0 & 0 & B_1 \Delta y^{-1}
\end{pmatrix}.
\]

\[
\mathbf{V}^R = [\varphi, \varphi_x \Delta x, \varphi_y \Delta y, \varphi_{xx}^R \Delta x^2, \varphi_{yy}^R \Delta y^2, \varphi_{xy}^R \Delta x \Delta y]^T,
\]

\[
\mathbf{C}^R = \begin{pmatrix}
B_1 & B_2 & B_3 & B_4 & B_5 & B_6 \\
0 & B_1 \Delta x^{-1} & 0 & B_2 \Delta x^{-1} & 0 & B_3 \Delta x^{-1} \\
0 & 0 & B_1 \Delta y^{-1} & 0 & B_2 \Delta y^{-1} & 0 & B_3 \Delta y^{-1}
\end{pmatrix}.
\]

This is a more efficient hyperbolic rDG method. It is based on the same $\mathbf{C}$ and $\mathbf{V}$ as DG($P_0P_1$)+DG($P_0$), and thus has the same mass matrices. However, a higher order polynomial $\mathbf{U}^R_h$ is used for computing the flux and source term, and thus it yields a more accurate solution. The higher order terms, i.e., $\varphi_{xx}^R \Delta x^2, \varphi_{yy}^R \Delta y^2, \varphi_{xy}^R \Delta x \Delta y$ are computed from $\varphi_x$ and $\varphi_y$ by using reconstruction schemes. For example, in this study, a hybrid least-squares scheme (LS) [24] and a variational reconstruction scheme (VR) [147] have been implemented to obtain higher moments. This scheme is compared with a conventional $P_1$ DG method in terms of the degrees of freedom, and achieves second-order accuracy for both the solution $\varphi$, and the derivatives $\varphi_x$ and $\varphi_y$, and third-order accuracy for $\varphi$ in the advection limit or on regular (or mildly-distorted irregular) grids. This is the class of hyperbolic-rDG schemes that reduces the cost of the DG methods; it is one of the main target schemes in the present work.

- **DG($P_0P_3$)+rDG($P_0P_2$) - Hyperbolic rDG**

\[
\mathbf{V} = \begin{pmatrix}
\varphi \\
\varphi_x \Delta x \\
\varphi_y \Delta y
\end{pmatrix},
\]
This is an even more efficient hyperbolic-rDG scheme. It is still based on the same $C$ and $V$ with three degrees of freedom, but achieves third-order accuracy and fourth-order in the advection limit. The higher order terms, i.e., $\phi_{\cdot R}^{c, x R} \Delta x^2, \phi_{\cdot R}^{c, y R} \Delta y^2, \phi_{\cdot R}^{c, x y R} \Delta x \Delta y, \phi_{\cdot R}^{c, x x x R}, \phi_{\cdot R}^{c, y y y R}, \phi_{\cdot R}^{c, x y y R}$ are computed from $\bar{\phi}_x$ and $\bar{\phi}_y$ by using reconstruction schemes. In this case, a quadratic reconstruction is required, which can be more efficiently performed by the variational method than the least-squares method. The DG($P_0 P_3$)+rDG($P_0 P_2$) scheme is much more efficient than a conventional $P_2$ DG method that requires six degrees of freedom to achieve third-order accuracy in the solution and only second-order accuracy in the derivatives. This scheme, therefore, greatly reduces the cost of the DG method, and as will be shown later, it is demonstrated to be a robust and accurate scheme.

Table 4.1 summarizes the types of schemes considered in this study. Expected order of accuracy is indicated separately for the advection term, the diffusion term, and the solution gradient. It shows clearly that the hyperbolic DG schemes of the class DG($P_0 P_{k+1}$)+DG($P_k$) [104] is one-order-lower accurate in the diffusion term than the DG($P_k$) scheme of the same DoFs. On the other hand, the hyperbolic rDG schemes of the class DG($P_0 P_{k+1}$)+rDG($P_k$) are more accurate than the DG scheme of the same DoFs. Moreover, the hyperbolic rDG scheme is even more accurate than a conventional DG with a larger number of DoFs: e.g.,
compare DG(P₀P₃)+rDG(P₀P₂) with DG(P₂). As will be demonstrated later, the hyperbolic rDG schemes achieve super convergence in some cases, and thus can be much more efficient than expected.

Table 4.1: Comparison of DG schemes and hyperbolic-DG/rDG schemes for expected order of accuracy.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>DoFs(1D/2D/3D)</th>
<th>Advection</th>
<th>Diffusion</th>
<th>Gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>DG(P₁)</td>
<td>2/3/4</td>
<td>2nd</td>
<td>2nd</td>
<td>1st</td>
</tr>
<tr>
<td>DG(P₀P₁)+DG(P₀)</td>
<td>2/3/4</td>
<td>2nd</td>
<td>1st</td>
<td>1st</td>
</tr>
<tr>
<td>DG(P₀P₂)+rDG(P₀P₁)</td>
<td>2/3/4</td>
<td>3rd</td>
<td>2nd</td>
<td>2nd</td>
</tr>
<tr>
<td>DG(P₀P₃)+rDG(P₀P₂)</td>
<td>2/3/4</td>
<td>4th</td>
<td>3rd</td>
<td>3rd</td>
</tr>
<tr>
<td>DG(P₂)</td>
<td>3/6/10</td>
<td>3rd</td>
<td>3rd</td>
<td>2nd</td>
</tr>
<tr>
<td>DG(P₀P₂)+DG(P₁)</td>
<td>3/6/10</td>
<td>3rd</td>
<td>2nd</td>
<td>2nd</td>
</tr>
<tr>
<td>DG(P₀P₃)+rDG(P₁P₂)</td>
<td>3/6/10</td>
<td>4th</td>
<td>3rd</td>
<td>3rd</td>
</tr>
<tr>
<td>DG(P₃)</td>
<td>4/10/20</td>
<td>4th</td>
<td>4th</td>
<td>3rd</td>
</tr>
<tr>
<td>DG(P₀P₃)+DG(P₂)</td>
<td>4/10/20</td>
<td>4th</td>
<td>3rd</td>
<td>3rd</td>
</tr>
</tbody>
</table>

4.4 Numerical Flux

Classically, the conventional DG would need two numerical flux schemes to solve the advection-diffusion equation. While DG methods are naturally developed for hyperbolic equations, the diffusive flux are not that straightforward or efficient. However, with the FOHS, the rDG method can use well-established methods for hyperbolic systems. In this work, the simplest upwind method is applied for the numerical flux across the interface:

\[
F_{ij} = \frac{1}{2}(F_L + F_R) \cdot n_{ij} - \frac{1}{2} |A_n|(U_R - U_L).
\]  

(4.49)

where the subscripts \(L\) and \(R\) indicate the face values of the polynomials in the cell \(i\) and \(j\) (i.e., the interior and exterior values), respectively, \(n_{ij} = (n_x, n_y)\) is the unit directed area vector.

The remaining issue is how to compute the absolute Jacobian matrix \(|A_n|\). Let us start with
the simplest case, i.e., the linear diffusion equation. The Jacobian matrix is given in Eq. 4.18. Thus, the absolute Jacobian $|A_n|$ is constructed by the right-eigenvector matrix $R_n$, and the diagonal eigenvalue-matrix $\Lambda_n$,

$$
R_n = \frac{1}{2} \begin{pmatrix}
1 & -1 & 0 \\
\frac{n_x}{\sqrt{\nu T_r}} & \frac{n_x}{\sqrt{\nu T_r}} & -2(D_{xx}n_x + D_{xy}n_y) \\
\frac{n_y}{\sqrt{\nu T_r}} & \frac{n_y}{\sqrt{\nu T_r}} & -2(D_{xy}n_x + D_{yy}n_y)
\end{pmatrix}, \Lambda_n = \begin{pmatrix}
-\lambda & 0 & 0 \\
0 & \lambda & 0 \\
0 & 0 & 0
\end{pmatrix},
$$

(4.50)

where

$$
\lambda = |\lambda_1| = |\lambda_2| = \sqrt{\nu / T_r}.
$$

(4.51)

Hence,

$$
|A_n| = R_n|\Lambda_n|R_n^{-1} = \lambda \begin{pmatrix}
1 & 0 & 0 \\
0 & n_x(D_{xx}n_x + D_{xy}n_y) & n_y(D_{xx}n_x + D_{xy}n_y) \\
0 & n_x(D_{xy}n_x + D_{yy}n_y) & n_y(D_{xy}n_x + D_{yy}n_y)
\end{pmatrix}.
$$

(4.52)

If the diffusion tensor $D$ is a simple constant, i.e.,

$$
D = \begin{pmatrix}
\nu & 0 \\
0 & \nu
\end{pmatrix},
$$

(4.53)

the absolute Jacobian would be simplified to

$$
|A_n| = \frac{\nu}{L_T} \begin{pmatrix}
1 & 0 & 0 \\
0 & n_x^2 + n_x n_y & n_x n_y \\
0 & n_x n_y & n_y^2
\end{pmatrix}.
$$

(4.54)

When it comes to the advection-diffusion equation, we would use an approximation following Refs. [115], which leads to

$$
F_{ij} = \frac{1}{2}(F_L + F_R) \cdot n_{ij} - \frac{1}{2}(|A_n^a| + |A_n^d|)(U_R - U_L),
$$

(4.55)

where $|A_n^a|$ and $|A_n^d|$ would be

$$
|A_n^a| = \begin{pmatrix}
|a_n| & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}, \quad |A_n^d| = \frac{\nu}{L_T} \begin{pmatrix}
1 & 0 & 0 \\
0 & n_x^2 & n_x n_y \\
0 & n_x n_y & n_y^2
\end{pmatrix}.
$$

(4.56)
Note that, the absolute Jacobian is constructed independently for both advection and diffusion terms. Here we are not assuming $|A_n| = |A_{nL}| + |A_{nN}|$, which is not true. What we did here is an approximation, which would allow us to avoid the analysis for the eigen-structure for the whole system. As a matter of fact, for this simple advection-diffusion equation, the eigen-structure for the whole system is still analyzable with some extra effort [110]. However, when it comes to complex conservation laws, such as the Navier-Stokes equations, one can only rely on the approximation approach at present. The simplified approach used here has been successfully demonstrated for the Navier-Stokes system [77, 79, 107, 111, 117].

As for the nonlinear diffusion equation, in which the positive diffusion coefficient $\nu$ is a function of the solution $\varphi$, the similar approximation would be applied, leading to

$$F_{ij} = \frac{1}{2}(F_L + F_R) \cdot n_{ij} - \frac{1}{2}(|A_{nL}^L| + |A_{nN}^N|)(U_R - U_L), \quad (4.57)$$

where

$$|A_{nL}^L| = \frac{\nu}{L_n} \begin{pmatrix} 1 & 0 & 0 \\ 0 & n_x^2 & n_x n_y \\ 0 & n_x n_y & n_y^2 \end{pmatrix}, \quad |A_{nN}^N| = \begin{pmatrix} \frac{\partial \nu}{\partial \varphi} |v_n n_k| & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \quad (4.58)$$

### 4.5 Boundary Condition

Boundary conditions are enforced weakly through the numerical flux in a similar manner as in the previous work [81, 120]. For all test problems, the Dirichlet condition is considered, and therefore only the solution $\varphi$ is given on boundaries. At a boundary face, $n_{ij}$ is taken to be outward, and thus $U_R$ is considered as a boundary state. The boundary condition is incorporated into the boundary state as

$$U_R = (\varphi_b, v_n n_x + \partial_s \varphi_b n_x, v_n n_y + \partial_s \varphi_b n_y), \quad (4.59)$$

where $\varphi_b$ is the value given as a boundary condition, and $\partial_s \varphi_b$ is the tangential derivative that can be obtained from the given boundary condition, $v_n$ is the face-normal projection of $(v_x, v_y)$ evaluated at the left (interior) state $U_L$, $n_{ij} = (n_x, n_y)$, and $(t_x, t_y)$ denotes a unit tangent vector of the boundary face. Note that $s$ is taken to be positive in the counterclockwise direction along a boundary, and the tangent vector is also taken in the same direction. In the
case of a unit square domain, the boundary state becomes

\[ \mathbf{U}_R = (\varphi_b, v_x, \partial_y \varphi_b), \]  

(4.60)

at the left and right boundaries, and

\[ \mathbf{U}_R = (\varphi_b, \partial_x \varphi_b, v_y), \]  

(4.61)

at the top and bottom boundaries.

Note that the normal component \( v_n \) may be specified in place of \( \varphi_b \) in the case of Neumann problems. As discussed in [109], the hyperbolic diffusion system has one wave going out of the domain, and therefore one quantity should be left unspecified, which corresponds to the normal derivative \( v_n \) in the Dirichlet case (or \( \varphi \) in the Neumann case). Or it may be argued that since the hyperbolic diffusion system is equivalent to the original diffusion equation in the pseudo steady state, the boundary condition should also be the same as the original problem. The tangential derivative can be specified since \( \varphi \) is known in the Dirichlet case, but it is not necessary; the results are very similar with and without specifying \( \partial_s \varphi_b \).
Temporal Integration Methods

The spatial discretization of the governing equations with the reconstruction-based discontinuous Galerkin method leads to a system of ordinary differential equations (ODEs) in time and

\[
M \frac{dU}{dt} = R(U^R),
\]

Eq. 3.16 can be written in an elemental semi-discrete form as

\[
M \frac{dU}{dt} = R(U^R),
\]

where \(U = (U_1, U_2, \ldots, U_k, \ldots, U_{Nelem})^T\) is the global solution vector of \(Ndegr \times Netot \times Nelem\) degrees of freedom to be evolved in time. Each \(U_k\) itself is a vector, which represents solution of \(Ndegr \times Netot\) degrees of freedom for the \(k\)-th element, for which the normalized linear (P1) solution vector can be written as follows:

\[
U_i = \left( \rho, \frac{\partial \rho}{\partial x} \Delta x, \frac{\partial \rho}{\partial y} \Delta y, \frac{\partial \rho}{\partial z} \Delta z, \frac{\partial \rho u}{\partial x} \Delta x, \frac{\partial \rho u}{\partial y} \Delta y, \frac{\partial \rho u}{\partial z} \Delta z, \frac{\partial \rho v}{\partial x} \Delta x, \frac{\partial \rho v}{\partial y} \Delta y, \frac{\partial \rho v}{\partial z} \Delta z, \frac{\partial \rho w}{\partial x} \Delta x, \frac{\partial \rho w}{\partial y} \Delta y, \frac{\partial \rho w}{\partial z} \Delta z, \frac{\partial \rho e}{\partial x} \Delta x, \frac{\partial \rho e}{\partial y} \Delta y, \frac{\partial \rho e}{\partial z} \Delta z \right).
\]

Likewise, \(R = (R_1, R_2, \ldots, R_k, \ldots, R_{Nelem})^T\) represents the global residual vector, where \(R_i\) is the elemental residual vector for the \(i\)-th element, and approaches zero for a steady-state solution. The global mass matrix \(M\) has a block-diagonal structure that couples the
\( N_{\text{deg}} \times N_{\text{tot}} \) degrees of freedom of each component of the solution vector only within \( \Omega_i \).

The \( i \)-th elemental mass matrix for the linear DG(P\(_{1}\)) method can be written as follows

\[
M_i = \begin{pmatrix}
\int_{\Omega_i} \tilde{B}_1 \tilde{B}_1 \, d\Omega & \int_{\Omega_i} \tilde{B}_1 \tilde{B}_2 \, d\Omega & \int_{\Omega_i} \tilde{B}_1 \tilde{B}_3 \, d\Omega & \int_{\Omega_i} \tilde{B}_1 \tilde{B}_4 \, d\Omega \\
\int_{\Omega_i} \tilde{B}_2 \tilde{B}_1 \, d\Omega & \int_{\Omega_i} \tilde{B}_2 \tilde{B}_2 \, d\Omega & \int_{\Omega_i} \tilde{B}_2 \tilde{B}_3 \, d\Omega & \int_{\Omega_i} \tilde{B}_2 \tilde{B}_4 \, d\Omega \\
\int_{\Omega_i} \tilde{B}_3 \tilde{B}_1 \, d\Omega & \int_{\Omega_i} \tilde{B}_3 \tilde{B}_2 \, d\Omega & \int_{\Omega_i} \tilde{B}_3 \tilde{B}_3 \, d\Omega & \int_{\Omega_i} \tilde{B}_3 \tilde{B}_4 \, d\Omega \\
\int_{\Omega_i} \tilde{B}_4 \tilde{B}_1 \, d\Omega & \int_{\Omega_i} \tilde{B}_4 \tilde{B}_2 \, d\Omega & \int_{\Omega_i} \tilde{B}_4 \tilde{B}_3 \, d\Omega & \int_{\Omega_i} \tilde{B}_4 \tilde{B}_4 \, d\Omega
\end{pmatrix}.
\] (5.3)

Due to its symmetry and Eq. 3.12, only the volume of the element and other six components \( \int_{\Omega_k} \tilde{B}_i \tilde{B}_j \, d\Omega \) \( (j \geq i) \) need to be actually pre-computed and stored, as shown in the following equation

\[
M_i = \begin{pmatrix}
V_i & 0 & 0 & 0 \\
0 & \int_{\Omega_i} \tilde{B}_2 \tilde{B}_2 \, d\Omega & \int_{\Omega_i} \tilde{B}_2 \tilde{B}_3 \, d\Omega & \int_{\Omega_i} \tilde{B}_2 \tilde{B}_4 \, d\Omega \\
0 & \int_{\Omega_i} \tilde{B}_3 \tilde{B}_2 \, d\Omega & \int_{\Omega_i} \tilde{B}_3 \tilde{B}_3 \, d\Omega & \int_{\Omega_i} \tilde{B}_3 \tilde{B}_4 \, d\Omega \\
0 & \int_{\Omega_i} \tilde{B}_4 \tilde{B}_2 \, d\Omega & \int_{\Omega_i} \tilde{B}_4 \tilde{B}_3 \, d\Omega & \int_{\Omega_i} \tilde{B}_4 \tilde{B}_4 \, d\Omega
\end{pmatrix}.
\] (5.4)

### 5.1 Explicit Scheme

#### 5.1.1 Three-stage Third-order TVD Runge-Kutta Scheme

The following explicit three-stage third-order TVD Runge-Kutta scheme (TVDRK3) \([27, 28]\)

\[
U^{(1)} = U^n + \Delta t M^{-1} R(U^n),
\]

\[
U^{(2)} = \frac{3}{4} U^n + \frac{1}{4} (U^{(1)} + \Delta t M^{-1} R(U^{(1)})),
\] (5.5)

\[
U^{n+1} = \frac{1}{3} U^n + \frac{2}{3} (U^{(2)} + \Delta t M^{-1} R(U^{(2)})).
\]

is widely used to advance the solution in time. This method linearly stable for a CFL number less than or equal to \(1/(2p+1)\).

#### 5.1.2 Four-stage Fourth-order Runge-Kutta Scheme

For some unsteady DNS problem, like Taylor-Green vortex problem in 3D, if explicit scheme is used, one may consider a higher order temporal approximation. Here, the most widely known member of the Runge-Kutta family, that is the four-stage fourth order Runge-Kutta scheme
(RK4) is used. It can be applied in a similar manner of TVDRK3.

\[
U^{(1)} = \frac{1}{2}U^n + \frac{1}{4}\Delta t M^{-1}R(U^n),
\]

\[
U^{(2)} = \frac{1}{3}U^n + \frac{1}{3}[U^{(1)} + \Delta t M^{-1}R(U^{(1)})],
\]

\[
U^{(3)} = \frac{1}{2}U^n + \frac{1}{2}[U^{(2)} + \Delta t M^{-1}R(U^{(2)})],
\]

\[
U^{n+1} = U^n + U^{(3)} + \Delta t M^{-1}R(U^{(3)}).
\]

\[\text{(5.6)}\]

### 5.2 Implicit Scheme

#### 5.2.1 Backward Euler Formulation

For steady state problems, the spatially discretized governing equations could be integrated implicitly in time, we can apply the backward Euler scheme to Eq. 5.1.

\[
M\left(\frac{U^{n+1} - U^n}{\Delta t}\right) = R(U^{n+1}).
\]

\[\text{(5.7)}\]

Since the governing equations are nonlinear, Eq. 5.7 is a system of nonlinear equations for the global solution vector \(U^{n+1}\). In order to solve this type of equations, we can linearize the right-hand-side vector \(R\) with respect to the global solution vector \(U\) at the current timestep

\[
R(U^{n+1}) \approx R(U^n) + \left(\frac{\partial R}{\partial U}\right)^n(U^{n+1} - U^n),
\]

\[\text{(5.8)}\]

where \(\left(\frac{\partial R}{\partial U}\right)^n\) is the so-called Jacobian matrix of the system evaluated at timestep \(n\), and denoted symbolically as \(J(U^n)\), which involves the linearization of both the inviscid and viscous flux functions.

If we plug Eq. 5.8 into Eq. 5.7 and move \(J(U^n)(U^{n+1} - U^n)\) to the left side, it then leads to a delta form of the equations

\[
A\Delta U^n = \left(\frac{M}{\Delta t} - \left(\frac{\partial R}{\partial U}\right)^n\right)\Delta U^n = R(U^n)
\]

\[\text{(5.9)}\]

where \(\Delta t\) is the time increment and \(\Delta U^n = U^{n+1} - U^n\) is the difference of global solution
vector between time level $n$ and $n + 1$. Note that if $\Delta t$ tends to infinity, the scheme reduces to the standard Newton’s method with a property of quadratic convergence for solving a system of nonlinear equations. Similarly, if we consider to compute the steady state solution of the discretized model equation of the hyperbolic reconstructed discontinuous Galerkin method, one can drop off the term associated with the physical time. Next, the pseudo-time integration towards the steady state could be performed by the BDF1 scheme. Note that the Jacobian matrix $J(U^n)$ is constant for the linear equations, and needs to be computed only once at the beginning of the computation.

5.2.2 Explicit First Stage, Single Diagonal Coefficient, Diagonally Implicit Runge-Kutta Scheme

For unsteady problems, the use of explicit time integration schemes has been widespread for DG discretization. While such methods are well suited for problems with similar spatial and temporal scales, they are notoriously inefficient for problems with disparate temporal and spatial scales, such as low-reduced frequency problems. In this work we focus on the use of implicit high-order time-integration schemes, which are not restricted by the CFL-stability limit of explicit methods, and thus are capable of using maximum time-steps determined by accuracy considerations, and are thus more suitable for stiff problems.

The first- and second-order BDF formulas are unconditionally stable (L-stable), and thus suitable for practical applications. However, higher-order ($\geq 3$) backward-difference formulas are only conditionally stable. Also, the high-order BDF method is not self-starting, requiring several lower order BDF methods at the starting stages. Furthermore, the time step would need to be fixed unless some further modification is made, like the variable time step BDF methods [120]. To overcome these difficulties, we consider an explicit first stage, single diagonal coefficient, diagonally implicit Runge-Kutta time integration scheme (ESDIRK) [19] and demonstrate the unsteady capability of the developed hyperbolic schemes. Compared with BDF methods, implicit Runge-Kutta (IRK) methods are A-stable and L-stable for arbitrary order in time. Also, variable time step sizes can be easily applied. Moreover, ESDIRK schemes are self-starting, i.e., one does not need to set up different temporal schemes at the beginning. Although ESIRK schemes would be more computationally expensive than the BDF counterpart for the same time step size, the cost can be reduced by taking a larger time step without encountering instability and maintaining the design order of accuracy. The ESDIRK schemes are highly desirable for practical applications, being unconditionally stable and thus allowing arbitrary size of time step without introducing low-order errors. The use of a lower-order time
integration scheme such as the BDF2 method may be allowed, but the time step needs to be small enough in order not to waste high-order spatial accuracy. To avoid such a possibility, the time-integration scheme must be stable and high-order, and the ESDIRK schemes are highly suitable to meet the requirement. In this study, we focus on the third-order ESDIRK scheme with a small time step and demonstrate that the developed hyperbolic rDG schemes can be used to solve unsteady problems with the ESDIRK scheme. It is noted that the main focus of the present study for the unsteady capability with high order implicit time schemes of the developed hyperbolic rDG scheme, since the ESDIRK has been successfully implemented in the rDG($P_nP_m$) framework for solving Euler/Navier-Stokes equations [76,154].

The following approaches manage to extend the ESDIRK method to the presented hyperbolic rDG method, with BDF1 discretization on the pseudo time derivatives. First, ESDIRK scheme for the physical time integration of Eq. 4.28 yields,

(i) \[ V^{(1)} = V^n, \]

(ii) For \( i = 2, \ldots, m, \)

\[
M_r \frac{\partial V^{(i)}}{\partial \tau} + \frac{M_l}{\Delta t} (V^{(i)} - V^n) = \sum_{j=1}^{i} a_{ij} R(V^{(j)}),
\]

(iii) \[ V^{n+1} = V^{(m)}, \]

where \( \Delta t \) is a physical time step, and \( a_{ij} \) are the Butcher coefficients of the scheme. The Butcher table for the third-order ESDIRK3 scheme \( (m = 4) \) employed can be found in Reference [19]. Due to the fact that \( a_{11} = 0 \), the first stage is explicit. At the \( i \)-th stage \( (i \geq 2) \), if one defines the source term associated with it as

\[
Q = \frac{M_l}{\Delta t} V^n + \sum_{j=1}^{i-1} a_{ij} R(V^{(j)}),
\]

the equation for the intermediate stages can be rewritten as

\[
M_r \frac{\partial V^{(i)}}{\partial \tau} = a_{ii} R(V^{(i)}) + Q - \frac{M_l}{\Delta t} V^{(i)}. \]

This system needs to be solved at each stage for the pseudo steady state. It is solved by the
steady solver described in the previous section, i.e.,
\[ V^{(i),k+1} = V^{(i),k} + \Delta V, \]  
where
\[ \left( \frac{M_t}{\Delta t} + \frac{M_t}{\Delta t} - a_{ii} \frac{\partial R}{\partial V} \right) \Delta V = a_{ii} R(V^{(i),k}) + Q - \frac{M_t}{\Delta t} V^{(i),k}. \]  

5.3 \( p \)-multigrid Method

Nowadays, geometric multigrid methods are very popular among CFD filed to accelerate the convergence of Euler and Navier-Stokes equations to a steady state on unstructured grids. Analysis of geometric multigrid indicates mesh that independent results are possible, which leads to drastically reduction of the computational cost. \( p \)-Multigrid method is a natural extension of geometric multigrid methods to high-order finite element formulation, such as spectral-\( hp \) or discontinuous Galerkin methods, where systems of equations are solved by recursively iterating on solution approximations of different polynomial order.

The basic idea of a \( p \)-multigrid method is to perform time steps on the lower order approximation levels to obtain the corrections to a solution on a higher order approximation level. Here, we would use rDG(P1P2) method for illustration. A two level V-cycle \( p \)-multigrid method has been used to drive the iterations. More specifically, this two level \( p \)-multigrid method consists of the following steps at each \( p \)-multigrid cycle:

1. Perform a time-step at the high approximation order, that is rDG(P1P2), which yields the initial solution \( U^{n+1}_{P1P2} \).

2. Transfer the flow solution and residual to the low approximation level, that is, DG(P0). This can be readily obtained using the shape function as
\[ U_{P0}(\Omega_e) = \sum_{i=1}^{N} U^{n+1}_{P1P2} B_i(x_e), \]  
\[ R_{P0}(\Omega_e) = \sum_{i=1}^{N} R^{n+1}_{P1P2} B_i(x_e), \]  
where \( x_e \) is the coordinates of the center of element \( \Omega_e \).

3. Compute the force terms on the lower approximation level
\[ F_{P0} = R_{P0} - R(U_{P0}). \]
(4) Perform a time-step at the lower approximation level where the residual is given by

\[ \mathbf{R} = \mathbf{R}(U^p_0) + \mathbf{F}_p_0. \]  

(5.18)

which yields the solution at the lower level \( U^{n+1}_{P_0} \).

(5) Interpolate the correction \( C_{P_0} \) back from the lower level to update the higher level solution

\[ C_{P_0} = U^{n+1}_{P_0} - U_{P_0}, \]  

(5.19)

\[ \bar{U}^{n+1}_{P_1 P_2} = U^{n+1}_{P_1 P_2} + C_{P_0}. \]  

(5.20)

As for time integration, \( p \)-multigrid method allows different schemes for different level approximation. In the present work, we would employ three-stage TVD Runge-Kutta explicit method due to the consideration of storage limitation. As a matter of fact, the storage limitation is one of the most important consideration when it comes to GPU computing. As for the lower level approximation, \( \text{DG}(P_0) \), where the storage requirement is not as demanding as in the higher level, first order backward implicit Euler time integration is utilized.

5.4 Solution of the Linear System of Equations

As shown earlier, both steady (BDF1) and unsteady problems (ESDIRK), need to solve the resulting linear system of equations. Therefore, in this section, we will give a detailed description as to how to solve the linear system of equations. We will continue this section based on the linear system of equations described by Eq. 5.7. The extension of solving the other linear system, e.g. Eq. 5.14 is straightforward.

The linear system of equations is at first left-multiplied by an inverse of the preconditioning matrix \( \mathbf{P} \)

\[ \mathbf{P}^{-1} \mathbf{A} \mathbf{U}^n = \mathbf{P}^{-1} \mathbf{R}(\mathbf{U}^n), \]  

(5.21)

where the matrix \( \mathbf{P} \) consists of the strict upper \( \mathbf{U} \), lower \( \mathbf{L} \) and diagonal \( \mathbf{D} \) matrices,

\[ \mathbf{P} = (\mathbf{D} + \mathbf{L}) \mathbf{D}^{-1} (\mathbf{D} + \mathbf{U}). \]  

(5.22)

Here \( \mathbf{L} \) and \( \mathbf{U} \) are stored via an face-based data structure. In order to construct an elemental Jacobian matrix for cell \( i \), the contributions from \( \mathbf{U}, \mathbf{L} \) and \( \mathbf{D} \) in face integrals are computed.
as below

\[ U = \int_{\Gamma_{ij}} \frac{\partial H_{\text{inv}}(U_i, U_j, n_{ij}) B_{d,ij}}{\partial U_j} \, d\Gamma - \int_{\Gamma_{ij}} \frac{\partial H_{\text{vis}}(U_i, U_j, n_{ij}) B_{d,ij}}{\partial U_j} \, d\Gamma, \tag{5.23} \]

\[ L = \int_{\Gamma_{ij}} \frac{\partial H_{\text{inv}}(U_i, U_j, n_{ij}) B_{d,ij}}{\partial U_i} \, d\Gamma + \int_{\Gamma_{ij}} \frac{\partial H_{\text{vis}}(U_i, U_j, n_{ij}) B_{d,ij}}{\partial U_i} \, d\Gamma, \tag{5.24} \]

\[ D_{\Gamma}^{i<j;i>j} = \sum_{\Gamma_{ij}} \left( \int_{\Gamma_{ij}} \frac{\partial H_{\text{inv}}(U_i, U_j, n_{ij}) B_{d,ij}}{\partial U_i} \, d\Gamma - \int_{\Gamma_{ij}} \frac{\partial H_{\text{vis}}(U_i, U_j, n_{ij}) B_{d,ij}}{\partial U_i} \, d\Gamma \right). \tag{5.25} \]

The contribution to \( D \) from domain integrals is

\[ D_{\Omega} = - \int_{\Omega} \frac{\partial F_k(U_i)}{\partial U_i} \frac{\partial B_{d,i}}{\partial x_k} \, d\Omega + \int_{\Omega} \frac{\partial G_k(U_i)}{\partial U_i} \frac{\partial B_{d,i}}{\partial x_k} \, d\Omega. \tag{5.26} \]

In Eq. 5.23, Eq. 5.24, Eq. 5.25 and Eq. 5.26, the subscript \( d \) denotes the index of DOFs, \( 1 \leq d \leq N_{\text{deg}} \); \( i \) and \( j \) denote cell \( i \) and its adjacent face-neighboring cell \( j \), respectively. Finally, the time derivative term \( M/\Delta t \) is added to the \( D \), and the elemental block diagonal matrix is as below

\[ D_i = \frac{M_i}{\Delta t} + D_{\Gamma_i} + D_{\Omega_i} = \frac{M_i}{\Delta t} - J_i \tag{5.27} \]

in which the crucial part is the assembly of the Jacobian matrix \( J_i \), and we will devote an effort to it in §5.4.1. The global block diagonal matrix requires a storage of \( N_{\text{elem}} \times (N_{\text{deg}} \times N_{\text{tot}})^2 \) units. Both the upper and lower matrices require a storage of \( N_{\text{fac}} \times (N_{\text{deg}} \times N_{\text{tot}})^2 \) units, where \( N_{\text{fac}} \) is the number of faces.

Clearly, the key point to solve the linear system lies in two aspect, that is how to obtain the Jacobian matrix and the choice of the linear solver. Hence, the next two subsections would give more details of discussion with these two aspect.

### 5.4.1 Jacobian Matrix

To formulate the Jacobian matrix, the exact linearization of flux functions can be considered. But the process can be quite complicated due to the fact that it involves the linearization of the reconstruction process, which is highly nonlinear and complex in nature and practically not accessible in a closed form, and the computing time and storage requirement would be prohibitively demanding. Thus, approximate Jacobians are usually formulated instead of the exact ones for the implicit finite volume / discontinuous Galerkin methods.
In order to make a balance between the speedup and extra computational cost by implementing an implicit algorithm, the Jacobian matrix is evaluated based on the underlying DG solution \((P_n)\) linearization \(J_{P_n}\) of the same order RHS operator \(R_{P_n}\), instead of the reconstructed higher-order RHS operator \(R_{P_nP_m}\). Eq. 5.9 can be rewritten in the form of approximate linear system as follows

\[
\left( \frac{M}{\Delta t} - J_{P_n} \right) \Delta U = R_{P_nP_m}.
\]  

(5.28)

Due to the inexact representation of the LHS matrix used in Eq. 5.28, the quadratic convergence of the Newton’s method can no longer be achieved in the case of \(n \neq m\). On the other side, the advantage of doing so lies in the fact that the implicit rDG(P\(_1\)P\(_2\)) method only requires a \(P_1\)-like linear system with significantly reduced memory requirement and computing time, while the scheme still remains to be third-order accurate in space. How to obtain the linearization (or differentiation) of a set of complicated functions, e.g., upwind numerical flux functions, is not trivial work. In general, the following four approaches are widely adopted in a variety of applications:

- **differentiation by hand**
  Manual implementation of analytic derivative formulae typically results in very efficient derivative code. However, the implementation is tedious and error-prone.

- **symbolic differentiation**
  Computer algebra packages manipulate expressions by repeatedly applying the chain rule so that there is no truncation error. However, the resulting expression for the derivative involves the parameters with respect to which one is differentiating. This can lead to an excessive growth of the length of the expression.

- **numerical differentiation**
  Divided differencing (DD) is based on some truncation of the Taylor series. It is easy to implement by evaluating the underlying function using perturbations of the input parameters. However, a suitable perturbation is often hard to find because a small perturbation decreasing the truncation error will increase the cancellation error.

- **automatic differentiation**
  Automatic differentiation (AD), also called algorithmic differentiation, is a technology for automatically augmenting computer programs, including arbitrarily complex simulations, with statements for the computation of derivatives, also known as sensitivities.
The Jacobian matrix is computed once per time step and used for either preconditioning or iteration solvers. In this work, we normally use the manually derived Jacobians for the Euler equations, and use the AD generated Jacobians for the Navier-Stokes equations, as they turned out to be most efficient for the inviscid and viscous flow problems respectively. As for the Jacobians of the model equations from hyperbolic rDG schemes, we have implemented both the exact Jacobians by hand, which is straightforward due to the simple hyperbolic formulation, and the AD versions. They would yield at the same Jacobian matrix, verifying our two approaches for the implementation.

5.4.1.1 Inviscid Flux

For the linearization of inviscid flux functions, we use an implicit HLLC flux scheme with the frozen acoustic wave-speed originally introduced by Batten et al. [17] for the finite volume methods, which demonstrated an un-compromised speed of convergence and robustness for smooth flows. In this work, we have extended this scheme into the discontinuous Galerkin space.

The implicit form (differentiation) of the HLLC flux functions in Eq. 3.40 are given by

$$
H^{n+1}_{\text{HLLC}}(U_l, U_r, n_{ij}) = \begin{cases} 
H^n_l + \frac{\partial H^n_l}{\partial U_l} \Delta U_l & \text{if } S_L > 0 \\
(H^n_l)^+ + \frac{\partial H^n_l}{\partial U_l} \Delta U_l + \frac{\partial H^n_r}{\partial U_r} \Delta U_r & \text{if } S_L \leq 0 < S_M \\
(H^n_r)^+ + \frac{\partial H^n_r}{\partial U_l} \Delta U_l + \frac{\partial H^n_r}{\partial U_r} \Delta U_r & \text{if } S_M \leq 0 \leq S_R \\
H^n_r + \frac{\partial H^n_r}{\partial U_r} \Delta U_r & \text{if } S_R < 0
\end{cases}, \quad (5.29)
$$

where for the supersonic case, $\partial H_l/\partial U_l$ and $\partial H_r/\partial U_r$ are linearization of the original flux functions, i.e., $\partial H/\partial U = \partial (F_x n_x + F_y n_y + F_z n_z) / \partial U$ and can be computed as follows:

$$
\frac{\partial F_x}{\partial U} = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
\varphi - u^2 & (3 - \gamma)u & (1 - \gamma)v & (1 - \gamma)w & \gamma - 1 \\
-wv & u & 0 & 0 & 0 \\
-wu & w & 0 & u & 0 \\
(\varphi - h)u & h - (\gamma - 1)u^2 & (1 - \gamma)uw & (1 - \gamma)uw & \gamma u
\end{pmatrix}, \quad (5.30)
$$
\[
\frac{\partial \mathbf{F}_y}{\partial \mathbf{U}} = \begin{pmatrix}
0 & 0 & 1 & 0 & 0 \\
-\varphi u & v & u & 0 & 0 \\
(\varphi - v^2) (1 - \gamma) u & (3 - \gamma) v & (1 - \gamma) w & \gamma - 1 \\
-\varphi w & 0 & w & v & 0 \\
(\varphi - h) v & (1 - \gamma) v u & h - (\gamma - 1) v^2 & (1 - \gamma) v w & \gamma v
\end{pmatrix}, \quad (5.31)
\]

and
\[
\frac{\partial \mathbf{F}_z}{\partial \mathbf{U}} = \begin{pmatrix}
0 & 0 & 0 & 1 & 0 \\
-wu & w & 0 & u & 0 \\
-wv & 0 & w & v & 0 \\
(\varphi - w^2) (1 - \gamma) u & (1 - \gamma) v & (3 - \gamma) w & \gamma - 1 \\
(\varphi - h) w & (1 - \gamma) w u & (1 - \gamma) w v & h - (\gamma - 1) w^2 & \gamma w
\end{pmatrix}. \quad (5.32)
\]

For the subsonic case, the HLLC Jacobian matrices are given by
\[
\frac{\partial \mathbf{H}_i^s}{\partial \mathbf{U}_i} = \begin{pmatrix}
(\frac{\partial \rho u_i^*}{\partial \mathbf{U}_i})^T S_M + (\frac{\partial S_M}{\partial \mathbf{U}_i})^T \rho_i^* \\
(\frac{\partial (\rho u)_i^*}{\partial \mathbf{U}_i})^T S_M + (\frac{\partial S_M}{\partial \mathbf{U}_i})^T (\rho u)_i^* + (\frac{\partial \rho u_i^*}{\partial \mathbf{U}_i})^T n_x \\
(\frac{\partial (\rho v)_i^*}{\partial \mathbf{U}_i})^T S_M + (\frac{\partial S_M}{\partial \mathbf{U}_i})^T (\rho v)_i^* + (\frac{\partial \rho v_i^*}{\partial \mathbf{U}_i})^T n_y \\
(\frac{\partial (\rho w)_i^*}{\partial \mathbf{U}_i})^T S_M + (\frac{\partial S_M}{\partial \mathbf{U}_i})^T (\rho w)_i^* + (\frac{\partial \rho w_i^*}{\partial \mathbf{U}_i})^T n_z \\
(\frac{\partial (\rho e)_i^*}{\partial \mathbf{U}_i} + \frac{\partial p_i^*}{\partial \mathbf{U}_i})^T S_M + (\frac{\partial S_M}{\partial \mathbf{U}_i})^T ((\rho e)_i^* + p^*)
\end{pmatrix}, \quad (5.33)
\]
and

\[
\frac{\partial H^*}{\partial \mathbf{U}_r} = \begin{pmatrix}
\left( \frac{\partial p_i^*}{\partial \mathbf{U}_r} \right)^T & S_M + \left( \frac{\partial S_M}{\partial \mathbf{U}_r} \right)^T \\
\left( \frac{\partial (\rho u)_i^*}{\partial \mathbf{U}_r} \right)^T & S_M + \left( \frac{\partial S_M}{\partial \mathbf{U}_r} \right)^T (\rho u)_i^* + \left( \frac{\partial \rho_i^*}{\partial \mathbf{U}_r} \right)^T n_x \\
\left( \frac{\partial (\rho v)_i^*}{\partial \mathbf{U}_r} \right)^T & S_M + \left( \frac{\partial S_M}{\partial \mathbf{U}_r} \right)^T (\rho v)_i^* + \left( \frac{\partial \rho_i^*}{\partial \mathbf{U}_r} \right)^T n_y \\
\left( \frac{\partial (\rho w)_i^*}{\partial \mathbf{U}_r} \right)^T & S_M + \left( \frac{\partial S_M}{\partial \mathbf{U}_r} \right)^T (\rho w)_i^* + \left( \frac{\partial \rho_i^*}{\partial \mathbf{U}_r} \right)^T n_z \\
\left( \frac{\partial (\rho e)_i^* + \rho_i^*}{\partial \mathbf{U}_r} \right)^T & S_M + \left( \frac{\partial S_M}{\partial \mathbf{U}_r} \right)^T ((\rho e)_i^* + p^*)
\end{pmatrix}
\]  

(5.34)

In Eq. 5.33 and Eq. 5.34, the vectors \( \frac{\partial S_M}{\partial \mathbf{U}_l}, \frac{\partial S_M}{\partial \mathbf{U}_r}, \frac{\partial p^*}{\partial \mathbf{U}_l}, \) and \( \frac{\partial p^*}{\partial \mathbf{U}_r} \) can be computed approximately. The \( S_M \) derivatives are computed as follows

\[
\frac{\partial S_M}{\partial \mathbf{U}_l} = \tilde{\rho}^{-1} \begin{pmatrix}
-q_l^2 + \psi_l(\gamma - 1)/2 + S_M S_L \\
n_x(2q_l - S_L - S_M) - (\gamma - 1)u_l \\
n_y(2q_l - S_L - S_M) - (\gamma - 1)v_l \\
n_z(2q_l - S_L - S_M) - (\gamma - 1)w_l \\
\gamma - 1
\end{pmatrix} ,
\]

(5.35)

\[
\frac{\partial S_M}{\partial \mathbf{U}_r} = \tilde{\rho}^{-1} \begin{pmatrix}
-q_r^2 - \psi_r(\gamma - 1)/2 - S_M S_R \\
n_x(-2q_r + S_R + S_M) + (\gamma - 1)u_r \\
n_y(-2q_r + S_R + S_M) + (\gamma - 1)v_r \\
n_z(-2q_r + S_R + S_M) + (\gamma - 1)w_r \\
-(\gamma - 1)
\end{pmatrix} ,
\]

(5.36)

where \( \psi \equiv u^2 + v^2 + w^2 \) and \( \tilde{\rho} \equiv \rho_r(S_R - q_r) - \rho_l(S_L - q_l) \). Taking differentiation of Eq. 3.46 with respect to \( \mathbf{U}_l \) and \( \mathbf{U}_r \) respectively gives

\[
\frac{\partial p^*}{\partial \mathbf{U}_l} = \rho_r(S_R - q_r) \frac{\partial S_M}{\partial \mathbf{U}_l} , \quad \frac{\partial p^*}{\partial \mathbf{U}_r} = \rho_l(S_L - q_l) \frac{\partial S_M}{\partial \mathbf{U}_r} .
\]

(5.37)

81
The remaining terms in Eq. 5.33 and Eq. 5.34 are given as follows

\[ \frac{\partial \rho_l^*}{\partial U_l} = \Omega_l \begin{pmatrix} S_L \\ -n_x \\ -n_y \\ -n_z \\ 0 \end{pmatrix} + \Omega_l \rho_l^* \frac{\partial S_M}{\partial U_l}. \] (5.38)

\[ \frac{\partial \rho_l^*}{\partial U_r} = \Omega_l \rho_l^* \frac{\partial S_M}{\partial U_r}. \] (5.39)

\[ \frac{\partial (p\nu)_l^*}{\partial U_l} = \Omega_l \begin{pmatrix} q_l u_l - n_x \psi_l (\gamma - 1)/2 \\ S_L - q_l + n_x (\gamma - 2) u_l \\ -u_l n_y + n_x (\gamma - 1) v_l \\ -u_l n_z + n_x (\gamma - 1) w_l \\ - (\gamma - 1) n_x \end{pmatrix} + \Omega_l \left( n_x \frac{\partial p^*}{\partial U_l} + (p\nu)_l^* \frac{\partial S_M}{\partial U_l} \right). \] (5.40)

\[ \frac{\partial (p\nu)_r^*}{\partial U_r} = \Omega_l \left( n_x \frac{\partial p^*}{\partial U_r} + (p\nu)_l^* \frac{\partial S_M}{\partial U_r} \right). \] (5.41)

\[ \frac{\partial (p\nu)_l^*}{\partial U_l} = \Omega_l \begin{pmatrix} q_l v_l - n_y \psi_l (\gamma - 1)/2 \\ -v_l n_x + n_y (\gamma - 1) u_l \\ S_L - q_l + n_y (\gamma - 2) v_l \\ -v_l n_z + n_y (\gamma - 1) w_l \\ - (\gamma - 1) n_y \end{pmatrix} + \Omega_l \left( n_y \frac{\partial p^*}{\partial U_l} + (p\nu)_l^* \frac{\partial S_M}{\partial U_l} \right). \] (5.42)

\[ \frac{\partial (p\nu)_r^*}{\partial U_r} = \Omega_l \left( n_y \frac{\partial p^*}{\partial U_r} + (p\nu)_l^* \frac{\partial S_M}{\partial U_r} \right). \] (5.43)

\[ \frac{\partial (p\nu)_l^*}{\partial U_l} = \Omega_l \begin{pmatrix} q_l w_l - n_z \psi_l (\gamma - 1)/2 \\ -w_l n_x + n_z (\gamma - 1) u_l \\ -w_l n_y + n_z (\gamma - 1) v_l \\ S_L - q_l + n_z (\gamma - 2) w_l \\ - (\gamma - 1) n_z \end{pmatrix} + \Omega_l \left( n_z \frac{\partial p^*}{\partial U_l} + (p\nu)_l^* \frac{\partial S_M}{\partial U_l} \right). \] (5.44)

\[ \frac{\partial (p\nu)_r^*}{\partial U_r} = \Omega_l \left( n_z \frac{\partial p^*}{\partial U_r} + (p\nu)_l^* \frac{\partial S_M}{\partial U_r} \right). \] (5.45)
\[
\frac{\partial (\rho e)_l^*}{\partial U_l} = \Omega_l \left( \left( \frac{(\rho e)_l + p_l}{\rho_l} q_l - q_l \psi_l (\gamma - 1)/2 \right) \right.
\]
\[
- n_x ((\rho e)_l + p_l)/\rho_l + (\gamma - 1) u q_l = \Omega_l \left( \frac{\partial p^*}{\partial U_l} S_M + (p^* + (\rho e)_l^*) \frac{\partial S_M}{\partial U_l} \right). \tag{5.46}
\]

\[
\frac{\partial (\rho e)_l^*}{\partial U_r} = \Omega_l \left( \frac{\partial p^*}{\partial U_r} S_M + (p^* + (\rho e)_l^*) \frac{\partial S_M}{\partial U_r} \right). \tag{5.47}
\]

In the case where \( S_M < 0 \), the relevant HLLC Jacobian matrices are obtained by simply interchanging subscript \( l \leftrightarrow r \) and \( L \leftrightarrow R \) in Eq. 5.38 through Eq. 5.47. This completes the definition of the frozen acoustic wavespeed version of the implicit HLLC flux. In addition, it was also found by Batten et al. \[17\] that very little speedup could be further achieved by computing the fully linearized implicit HLLC flux, due to the fact that the extra work required to compute the Jacobians including the differentiations of the acoustic wavespeed does not significantly favor this version over the approximated form.

Instead of evaluating the Jacobians based on the flux scheme in the RHS operator which is usually quite complicated, a much more simplified approach is also available for differentiation by hand. A fast matrix-free implicit finite volume method for compressible flows on unstructured grids was introduced by Luo et al. \[84, 96\], in which the Jacobian matrix is computed based on a simple spectral-type flux function, i.e., the Lax-Friedrichs scheme instead of the one used in the RHS operator. It was proved to be highly cost-effective in the finite volume context by carrying out numerical test cases on complex geometric configurations \[84, 96\]. This approach is named as the unmatching Jacobians (UJ) in this work, which is to distinguish from the RHS flux-based matched Jacobians (MJ), i.e., HLLC Jacobians. We have also extended this approach into the discontinuous Galerkin space to construct the elemental Jacobian matrix. The contributions from upper matrix \( U \), lower matrix \( L \) and diagonal matrix \( D \) in face integrals are computed as below

\[
U = \int_{\Gamma_{ij}} \frac{1}{2} \left( \frac{\partial F_k}{\partial U_j} n_k B_{d,ij} - |\lambda_{tr} I_{Netot \times Netot} B_{d,ij}| \right) d\Gamma, \tag{5.48}
\]

\[
L = \int_{\Gamma_{ij}} \frac{1}{2} \left( \frac{\partial F_k}{\partial U_i} n_k B_{d,ij} + |\lambda_{tr} I_{Netot \times Netot} B_{d,ij}| \right) d\Gamma, \tag{5.49}
\]

\[
D \Gamma = \sum_{\Gamma_{ij}} \int_{\Gamma_{ij}} \frac{1}{2} \left( \frac{\partial F_k}{\partial U_i} n_k B_{d,ij} + |\lambda_{tr} I_{Netot \times Netot} B_{d,ij}| \right) d\Gamma, \tag{5.50}
\]

83
where the spectral radii $|\lambda_{lr}|$ on the face integration point is given by

$$|\lambda_{lr}| = \frac{1}{2}(|(V_l + V_r) \cdot n_{ij}| + a_l + a_r),$$  \hfill (5.51)

where $a$ denotes the local speed of sound.

A matrix-free approach [84, 96] is used along with the unmatching Jacobians (UJ) for the lower (forward) and upper (backward) sweep of preconditioning, where the product of Jacobian matrix and incremental state variable vector is approximated by the increment of flux vector. The major advantage of doing so is that no static memory is required for storing the lower and upper matrices at all, which results in a significant reduction in memory requirement. Rasetariner and Hussaini [132] extended this matrix-free implicit scheme into the discontinuous Galerkin space for compressible inviscid flows on 2D structured grids. Nevertheless, it still remains to be explored whether the unmatching Jacobians are able to achieve equivalent or better performance than the matched Jacobians for compressible flows in 3D.

### 5.4.1.2 Viscous Flux

However, unlike the inviscid part, it is not straightforward to linearize the viscous flux resulting from BR2 exactly. Therefore, we turn into taking the advantages of auto differencing package with the treatment of freezing the global lift operator in the viscous domain integral. It is obvious that this approximation could greatly reduce the complexity of linearizing the viscous domain integral with respect to the solution vector $U$.

Automatic Differentiation (AD) is a set of techniques based on the mechanical application of the chain rule to obtain derivatives of a function given as a computer program. It has been used in many areas, including numerical methods, sensitivity analysis, design optimization, data assimilation and inverse problems. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations such as additions or elementary functions such as $\exp()$. By applying the chain rule of derivative calculus repeatedly to these operations, derivatives of arbitrary order can be computed automatically, and accurate to working precision.

There are usually two modes in an AD tool: forward mode and reverse mode. The derivatives of dependent with respect to independent variables computed by these modes are mathematically equivalent, but the time and memory requirements of computing them may differ. The forward mode computes derivatives of intermediate variables with respect to independent variables and propagates from one statement to the next according to the chain rule. The reverse mode computes derivatives of dependent variables with respect to intermediate variables and
propagates from one statement to the previous statement according according to the chain rule. Thus, the reverse mode requires a reversal of the program execution. The forward mode is sometimes referred to as direct or tangent linear mode whereas the reverse mode is also called backward, adjoint or cotangent linear mode.

Implementations of AD can be broadly classified into two categories. AD tools based on operator overloading exploit the fact that modern programming languages offer the possibility to redefine the semantics of elementary operators. AD tools based on source-to-source transformation change the semantics by explicitly rewriting the code. Each of these approaches has its advantages and disadvantages [20].

In this work, we have applied automatic differentiation to evaluating the flux Jacobian matrix. TAPENADE is a source-to-source AD tool [1]. Given a FORTRAN77, FORTRAN95, or C source program, it generates its derivative in forward (tangent) or reverse (adjoint) mode. TAPENADE is directly accessible through a web server, or can be downloaded locally, between which it is preferred in our work to obtaining the source code returned by the web server. Although there is no algorithmic difference between the source codes from the web server and a locally installed package, there are some advantages in sticking to the web server. First, one does not need to install, and second, one does not need to re-install to get updates of the tool.

A simple hand-made example is presented to demonstrate how TAPENADE web server generates the differential of a function. First, a FORTRAN77 file test.f which contains the routine func as shown in Table 5.1 is uploaded as a source file. Multiple source files or header files can also be provided. Then, the name of the top routine func, the dependent output variable array \( p_y \), and the independent input variable array \( p_x \) are specified, and the differentiation mode *Tangent Multidirectional Mode* is chosen.

<table>
<thead>
<tr>
<th>Table 5.1: A simple source program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>!... This routine demonstrates a simple algebraic function</td>
</tr>
<tr>
<td>SUBROUTINE FUNC(p_x, p_y)</td>
</tr>
<tr>
<td>IMPLICIT NONE</td>
</tr>
<tr>
<td>REAL*8 p_x(2), p_y(2)</td>
</tr>
<tr>
<td>p_y(1) = 2.0<em>p_x(1) + 3.0</em>p_x(2)</td>
</tr>
<tr>
<td>p_y(2) = 4.0*p_x(1)*p_x(2)</td>
</tr>
<tr>
<td>RETURN</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

TAPENADE will generate a file *test.dv.f* which contains the differentiation routine *func_dv*.
as shown in Table 5.2. There are two more inputs \( p_{xd}(nbdirsmx, 2) \) and \( nbdirs \), and one more output \( p_{yd}(nbdirsmx, 2) \) for the new routine. The \( nbdirsmx \) is a user-defined integer in \textit{DIFFSIZES.inc} as a header file. Like all the other AD tools, TAPENADE does not directly provide the routine for Jacobian matrix evaluation. Instead, it can provide the Jacobian-vector products based on user’s inputs. In order to obtain the Jacobian matrix, \textit{Tangent Multidirectional Mode} serves the purpose of obtaining the whole Jacobian matrix, or arbitrary columns in the matrix. The variable \( nbdirs \) is the number of directions that the user want to do loops over the Jacobian-vector products process. It is set equal to the subscript size of the vector \( p.x \). If \( p.xd \) is set equal to the identity matrix, then the computed matrix \( p.yd \) is just the transpose of the Jacobian matrix. The user can transpose \( p.yd \) back to obtain the Jacobian matrix.

Table 5.2: A simple differential source program.

```fortran

!... Differentiation of func in forward (tangent) mode: (multi-directional mode)
!... variations of useful results: p.y
!... with respect to varying inputs: p.x
!... RW status of diff variables: p.x:in p.y:out
SUBROUTINE FUNC_DV(p.x, p.xd, p.y, p.yd, nbdirs)
IMPLICIT NONE
INCLUDE 'DIFFSIZES.inc'
REAL*8 p.x(2), p.y(2)
REAL*8 p.xd(nbdirsmax, 2), p.yd(nbdirsmax, 2)
INTEGER nd
INTEGER nbdirs
INTEGER ii1
DO nd = 1, nbdirs
DO ii1 = 1, 2
p.yd(nd, ii1) = 0.0
ENDDO
p.yd(nd, 1) = 2.0*p.xd(nd, 1) + 3.0*p.xd(nd, 2)
p.yd(nd, 2) = 4.0*(p.xd(nd, 1)*p.x(2)+p.x(1)*p.xd(nd, 2))
ENDDO
p.y(1) = 2.0*p.x(1) + 3.0*p.x(2)
p.y(2) = 4.0*p.x(1)*p.x(2)
RETURN
END
```

Recall that a face-based data structure is used to store the lower matrix \( L \) and the upper matrix \( U \), and we still need to preserve this structure when the flux Jacobian matrix is evaluated by the AD tool. In order to make the least code modification, only the manually derived routines of Jacobian matrix evaluation are replaced with the corresponding AD generated source routines. Table 5.3 presents a series of exemplary procedures to implement TAPENADE.
automatic differentiation on the source routine \( rhsbp1 \), whose functionality is to compute the contribution of integrals to the left and right elemental RHS vectors across an interior face. The AD generated source code \( rhsbp1\_dv \) is then integrated in RDGFLO to compute the elemental Jacobian matrices for the left and right cells, and also the corresponding lower and upper matrices, as outlined in Table 5.4.

Table 5.3: Example: implementation of TAPENADE on RDGFLO.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>User’s inputs or choice</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>▷ Select the input language</td>
<td>Fortran 95</td>
<td>RDGFLO is written in Fortran95</td>
</tr>
<tr>
<td>▷ Upload source code</td>
<td>top.F</td>
<td>It contains the top routine to be differentiated</td>
</tr>
<tr>
<td>▷ Name of the top routine (given)</td>
<td>rhsbp1_</td>
<td>Integrals of RHS over a face</td>
</tr>
<tr>
<td>▷ Dependent output variables for the given routine</td>
<td>( p_{rzl}, p_{rzn} )</td>
<td>Local elemental RHS vectors for left and right cells across a face</td>
</tr>
<tr>
<td>▷ Independent input variables for the given routine</td>
<td>( p_{unl}, p_{unr} )</td>
<td>Local elemental state vectors for left and right cells across a face</td>
</tr>
<tr>
<td>▷ Differentiate in</td>
<td>Tangent Multidirectional Mode</td>
<td></td>
</tr>
<tr>
<td>▷ Generated source code (returned)</td>
<td>( rhsbp1_dv_ )</td>
<td>Differentials of RHS over a face</td>
</tr>
<tr>
<td>▷ Dependent output variables for the returned routine</td>
<td>( p_{rzl}, p_{rzn}, p_{rzld}, p_{rzd} )</td>
<td></td>
</tr>
<tr>
<td>▷ Independent input variables for the returned routine</td>
<td>( p_{unl}, p_{unr}, p_{unld}, p_{unrd}, nbdirs )</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.4: Pseudo code: contribution of face integrals for the block diagonal matrix $D$, lower matrix $L$, and upper matrix $U$.

```fortran
subroutine adlhsbounp1(..., unkno, diago, lower, upper)
  implicit none
  real*8, dimension(Mdegr, Netot, Nsize) : unkno
  real*8, dimension(Ndofe, Ndofe, Nelem) : diago
  real*8, dimension(Ndofe, Ndofe, Nafac) : lower, upper
  ! local input arrays
  real*8, dimension(Ndofe) : p_unl, p_unr, p_rzl, p_rzr
  real*8, dimension(Ndofe, Ndofe, Nafac) : p_unld, p_unrd, p_rzld, p_rzrd
  !
  ! loop over the interior faces
  do ifa = Nbfac+1, Nafac
    !
    ! conservative variables at the left & right cells
    p_unl(1 : Ndofe) := unkno(1 : Ndegr, 1 : Netot, iel)
    p_unr(1 : Ndofe) := unkno(1 : Ndegr, 1 : Netot, ier)
    ! initialize the local RHS vectors
    p_rzl(1 : Ndofe) := 0.0 ; p_rzr(1 : Ndofe) := 0.0
    !
    ! initialize the identity matrix & number of directions for diago at left cell and lower matrix
    p_unld = 0.0 ; p_unrd = 0.0
    do ii = 1, Ndofe ; p_unld(ii, ii) = 1.0 ; enddo
    nbdirs = Ndofe
    ! differentiation of the local face integral
    call rhsb1_dv(..., p_unl, p_unld, p_unr, p_unrd, p_rzl, p_rzld, p_rzr, p_rzrd, nbdirs)
    ! transpose and get the Jacobian matrix at left cell, scatter to diago & lower
    if(iel <= nelem) diago(:, :, iel) = diago(:, :, iel) + transpose(p_rzld(:, :))
    lower(:, :, ifa) = lower(:, :, ifa) - transpose(p_rzld(:, :))
  
    ! initialize the identity matrix & number of directions for diago at right cell and upper matrix
    p_unld = 0.0
    do ii = 1, Ndofe ; p_unrd(ii, ii) = 1.0 ; enddo
    ! differentiation of the local face integral
    call rhsb1_dv(..., p_unl, p_unld, p_unr, p_unrd, p_rzl, p_rzld, p_rzr, p_rzrd, nbdirs)
    ! transpose and get the Jacobian matrix at right cell, scatter to diago & upper
    if(ier <= nelem) diago(:, :, ier) = diago(:, :, ier) - transpose(p_rzld(:, :))
    upper(:, :, ifa) = upper(:, :, ifa) + transpose(p_rzld(:, :))
  
  ! end of loop over the interior faces
  enddo
  !
  return
end
```

88
5.4.2 Linear Solver

In our framework, a linear system of equations can be solved iteratively by using either the Symmetric Gauss-Seidel (SGS) method, or the Generalized Minimal Residual (GMRES) method developed by Saad and Schulz [135] with LU-SGS preconditioning [84,96], namely GMRES+LU-SGS.

Since GMRES is among the most popular and efficient implicit algorithms in CFD [13,14,15,27,35,45,57,84,87,89,90,91,96,132,152], GMRES is one of the most robust Krylov methods, but it is also highly expensive in terms of storage, because all the computed vectors in the orthogonal sequence need to be retained. In order to alleviate the memory requirements, a restarted sub-version denoted as GMRES(m) is used, where after a user specified amount of m iterations, the stored data is cleared and the intermediate data is used as the initial solution for the next m iterations. There is no definite rules for the choice of m. Usually, we choose m = 10 as a rule of thumb. The structure of the GMRES(m) algorithm is presented in Table 5.5 to solve the linear system of equations within each time step. The GMRES algorithm can be partly matrix-free, for a finite difference approach can be used to compute the matrix-vector products instead of calculating and storing the full matrix as shown below

\[
\frac{\partial \mathbf{R}(\mathbf{U})}{\partial \mathbf{U}} \Delta \mathbf{U} \approx \frac{\mathbf{R}(\mathbf{U} + \varepsilon \cdot \Delta \mathbf{U}) - \mathbf{R}(\mathbf{U})}{\varepsilon},
\]

(5.52)

where the residual vector \( \mathbf{R}(\mathbf{U} + \varepsilon \cdot \Delta \mathbf{U}) \) of the governing equations is computed with the state variable vector \( \mathbf{U} \) added by a set of perturbed state quantities. The scalar parameter \( \varepsilon \) can be computed according to the formula proposed by Pernice and Walker [125]

\[
\varepsilon = \epsilon \sqrt{\frac{1 + \|\mathbf{U}\|_2^2}{\|\Delta \mathbf{U}\|_2}},
\]

(5.53)

where the parameter \( \epsilon \) is a user-defined input and the choice of this parameter is between the round-off and truncation error. In our applications, \( \varepsilon \) is used for all the computations without problem. The GMRES algorithm requires one flux evaluation per time step and one flux evaluation per inner GMRES iteration. The primary storage is dictated by the LU-SGS preconditioning, which requires the upper, lower and diagonal matrices to be stored for every non-zero element in the approximate left-hand-side matrix in Eq. 5.9. The additional storage associated to the GMRES algorithm is an array of size \((k + 2) \times \text{Nelem} \times (\text{Ndegr} \times \text{Netot})\), where \( k \) is the number of search directions. Due to the fact that GMRES requires additional vector storage, which is limited on GPU platform, we have not ported the GMRES solver onto
Table 5.5: Flowchart of the GMRES algorithm.

<table>
<thead>
<tr>
<th>DO $l = 1, m$</th>
<th>$m$ restarted iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0 = R - AU_0$</td>
<td>initial residual</td>
</tr>
<tr>
<td>$r_0 := P^{-1}v_0$</td>
<td>preconditioning step</td>
</tr>
<tr>
<td>$\beta := |r_0|_2$</td>
<td>initial residual norm</td>
</tr>
<tr>
<td>$v_1 := r_0/\beta$</td>
<td>define initial Krylov</td>
</tr>
<tr>
<td>DO $j = 1, k$</td>
<td>inner iteration</td>
</tr>
<tr>
<td>$y_j := \frac{\partial R(U)}{\partial U} \Delta U$</td>
<td>matrix-vector product</td>
</tr>
<tr>
<td>$w_j = P^{-1}y_j$</td>
<td>preconditioning step</td>
</tr>
<tr>
<td>DO $i = 1, j$</td>
<td>Gram-Schmidt step</td>
</tr>
<tr>
<td>$h_{i,j} = (w_j, v_i)$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$w_j = w_j - h_{i,j}v_i$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>END DO</td>
<td></td>
</tr>
<tr>
<td>$h_{j+1,j} := |w_j|_2$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$v_{j+1} := w_j/h_{j+1,j}$</td>
<td>define Krylov vector</td>
</tr>
<tr>
<td>END DO</td>
<td></td>
</tr>
<tr>
<td>$z := \min_z |\beta e - H\hat{z}|_2$</td>
<td>solve least squares</td>
</tr>
<tr>
<td>$\Delta U := \Delta U_0 + \sum_{i=1}^m v_i z_i$</td>
<td>approximate solution</td>
</tr>
<tr>
<td>IF $|\beta e - H\hat{z}|_2 &lt; \epsilon$ EXIT</td>
<td>convergence check</td>
</tr>
<tr>
<td>$\Delta U_0 := \Delta U$</td>
<td>restart</td>
</tr>
<tr>
<td>END DO</td>
<td></td>
</tr>
</tbody>
</table>

the GPU platforms yet. For the GPU accelerated implicit computation, SGS method is applied. As for the CPU computation, we stick to using the GMRES+LU-SGS scheme throughout the numerical test cases if implicit time advancement is required.
Chapter 6

OpenACC Based Parallelization

6.1 Overview

Graphics processing units, which are massively parallel platforms, have become useful in computational fluid dynamics solvers. Due to the compactness of the DG and also rDG methods makes them ideally suited for the implementation of parallel computing on regular clusters and supercomputers. However, the capabilities of the traditional CPU(central processing unit) based parallel computing may not meet the needs of solving complex simulation problems. Fortunately, the General-Purpose Graphics Processing Unit (GPGPU [123]) technology offers a new opportunity to significantly accelerate CFD simulations by offloading compute-intensive portions of the application to the GPU, while the remainder of the computer program still runs on the CPU, which also make it expected to be a major compute unit in the near future. For people who have experience with OpenMP, they would find OpenACC is much like OpenMP. What developers need to do is simply annotate their code to identify the areas that should be accelerated by wrapping with the OpenACC directives and some runtime library routines, instead of taking the huge effort to change the original algorithms as to accommodate the code to a specific GPU architecture and compiler. In that case, people benefit not only from easy implementation of the directives but also the freedom to compile the very same code and conduct computations on either CPU or GPU from different vendors, e.g., NVIDIA, AMD and Intel accelerators. Nevertheless, as for some cutting-edge features, OpenACC still lags behind CUDA due to vendors’ distribution plan (note that Nvidia is among the OpenACC’s main supporters). But still, OpenACC manages to offer a promising approach to minimize the effort to extend the existing legacy CFD codes while maintaining multi-platform and multi-vendor
compatibility, and thus to become an attractive parallel programming model.

As a matter of fact, the way to implement OpenACC is very similar to that of OpenMP. In order to achieve a competent speedup, the OpenACC parallel construct directives need to be properly inserted in the code for the compiler to generate the acceleration kernels. The example shown in Table 6.1 demonstrates the parallelization of a loop over the elements for collecting contribution to the residual vectors.

GPUs store data in shared memory fashion and manipulate them by processes that solve the problem in parallel. Hence, a race condition is created and computations if multiple processes, or threads, write simultaneously to the same memory location, leading to unpredictable results. Therefore, care must be taken to develop algorithms that do not lead to race conditions.

Typically, a GPU has hundreds or even thousands of computation cores. However, compared with a typical CPU core, the one on GPU card would has much less computation power and local cache memory. Therefore, for optimal performance on GPU, the algorithm should be divided into smaller units, thus occupying more cores with the same amount of work. In addition, the amount of data each core processes should be kept as small as possible. Therefore, the fine granularity algorithm, with the two above-mentioned benefits, is able to carry out high computation efficiency on GPGPU. Note that the latter aspect is particularly important for solving a block-sparse system, which consists of a large amount of square sub-matrices of an identical size. On each sub-matrix, if the operation is mapped to one GPU core, the algorithm easily becomes heavily memory-bounded, especially when the dimension of the sub-matrix is large, which is the common case for high order method, resulting in serious performance penalty.
In other words, the key to achieve higher speed up factor on GPU platform is fine granularity. On the other hand, fine granularity usually introduces more synchronization and memory access overhead. Compared with other open GPGPU programming model like OpenCL, OpenACC would suffer a more expensive overhead from kernel launches. Thus, the algorithm designers need to weigh costs and benefits to make a balanced choice.

Nevertheless, it is generally difficult to port the implicit algorithm to GPU platform. First of all, the size of sub-matrices would cause the memory-bounded issue, especially for higher order method. Secondly, it is not straightforward to employ some popular linear solver or preconditioner like Symmetric Gauss-Seidel (SGS) or Lower Upper-Symmetric Gauss-Seidel (LU-SGS) due to the inherent data dependency. Additionally, the iterative solver like Generalized Minimal Residual (GMRES) method would require additional storage for some auxiliary arrays, which would bring challenge to GPU computing since the local cache memory of GPGPU is limited.

In order to guarantee the maximal portability of the code and flexibility for parallel programming, the RDGFLO framework utilizes the OpenACC and Message Passing Interface (MPI) library for multiple GPUs parallel computation.

### 6.2 Face Coloring Method

As mentioned in the previous section, attention need to be paid for memory contention (race condition) for vectorizing CFD solvers on GPUs. No matter which temporal scheme is used for simulation, one would need to accumulate the residual vector in DG/rDG solver that consists of boundary integral over the faces and volumetric integral over the elements. The example shown in Table 6.1 is the domain integral over the elements using Gauss quadrature method. Both the OpenMP and OpenACC parallel construct directives can be applied to a readily vectorizable loop like that, without the need to modify the original code structure. However due to the unstructured grid topology, the attempt to directly wrap a loop over the dual edges for collecting contribution to the residual vector with either the OpenMP or the OpenACC directives would cause memory contention issue, in other words, multiple writes to the same elemental residual vector and thus resulting incorrect values. Unlike in the structured CFD solvers [101, 128], this kind of race condition issue is typically associated to the threading of dual edge loops in unstructured CFD solvers for both node-centered and the cell-centered schemes, which is not strange at all to those who have the experience of developing parallel unstructured CFD solvers based on OpenMP. Such race condition in the dual-edge loops can be avoid in different approaches.

First one is to incorporated the face integral into a grand loop over the elements as proposed
by Corrigan et al. \cite{34} for the FV methods. As a result, all the workload-intensive computations are wrapped in element-wise loops, which are perfect for threading. Thus no “race condition” would occur for the resulting code. However, a major overhead associated to this approach is its redundant computation for the dual-edges. According to Reference \cite{34}, the performance of the developed finite volume solver based on CUDA was only advantageous in single-precision GPU computation, and became much worse in double-precision. In fact, this approach would never meet our design goals for two reasons. For the first, the discontinuous Galerkin methods require an inner loop over the Gauss quadrature points for computing the face integrals in dual-edge computation, which account for at least 50% of the gross computing time as in the second-order DG(P_1) method. Furthermore, the number of required Gaussian quadrature points could be a larger number in the case of higher-order DG methods, therefore resulting in a huge overhead if the workload of such computation is doubled. For the second, the implementation of this approach indicates a major change in the code structure, which is not only costly in programming, but also would completely ruin the performance of the equivalent CPU code.

Alternatively, the “race condition” can be eliminated with a moderate amount of work by adopting a mature algorithm of face renumbering and grouping. This algorithm is designed to divide all the faces into a number of groups by ensuring that any two faces that belong to a common element never fall in the same group, so that the face loop in each group can be vectorized without “race condition”. An example is shown in Table 6.2, where an extra do-construct that loops over these groups is nested on top of the original loop over the internal faces, and executed sequentially. The inner do-construct that loops over the internal faces is vectorized without the “race condition” issue. In fact, this kind of algorithm is widely used for vectorized computing on unstructured grids with OpenMP. The implementation details can be found in an abundance of literature \cite{80,82,83,155,156,157}. The number of groups for each subdomain grid is usually between 6 and 8 according to a wide range of test cases, indicating some overheads in repeatedly launching and terminating the OpenACC acceleration kernels for the loop over the face groups. This kind of overheads is typically associated to GPU computing, but not for the code if parallelized by OpenMP for CPU computing. Nevertheless, the most favorable feature in this design approach is that it allows the original CPU code to be recovered when the OpenACC directives are dismissed in the pre-processing stage of compilation. Therefore, the use of this face renumbering and grouping algorithm will result in a unified source code for both the CPU and GPU computing on unstructured grids.

To sum up from the discussion above, the second parallelism strategy can suit well in the present work, for its simplicity and portability to quickly adapt into the original source code without any major change in the legacy programming structures.
In our work, a simple greedy coloring algorithm will loop through all surfaces in the mesh. On each edge, the face colors of its left and right element are checked. Then, the first available color is assigned to the surface. This greedy algorithm will yield meshes of a maximum five colors for triangular elements, and a maximum of seven colors for quadrilateral and tetrahedral elements. As Giuliani et al. [49] pointed out, this is not the minimum number of colors possible for these element geometries. An optimal number of colors for a variety of element geometries could be obtained by using an edge coloring heuristic, resulting a reduced number of kernel launches and streamlining of the code. The overhead associated with launching extra kernels is minor for GPUs, but non-negligible for multi-core CPUs [155]. However, in our case, the coloring method is only turn on when GPU is used for parallel computation. The multi-core CPUs parallelization for our RDGFLO solver utilized the Message Passing Interface (MPI) library and is designed for architectures with distributed memory instead of the share memory. Therefore, if the computation is performed only on CPU platform, the coloring scheme would be turned off to reduce the overheads.

This strategy is applied for the face integrals as well as some other procedures that require the loop over faces like $P_1P_2$ least-squares reconstruction and evaluation of the local time-step sizes.
Table 6.2: An example of loop over the edges.

```plaintext
!! OpenMP for CPUs (without race condition):
!! loop over the groups
Nfac1 = Njfac
do ipass = 1, Npass_ift
   Nfac0 = Nfac1 + 1
   Nfac1 = fpass_ift(ipass)

!! loop over the edges
!$omp parallel
!$omp do
   do ifa = Nfac0, Nfac1
      !! left element
      iel = intfac(1,ifa)

      !! right element
      ier = intfac(2,ifa)

      !! loop over Gauss quadrature points
      do ig = 1, Ngp
         !! contribution to the left element
         rhsel(*,*,iel) = rhsel(*,*,iel) - flux

         !! contribution to the right element
         rhsel(*,*,ier) = rhsel(*,*,ier) + flux
      enddo
   enddo
!$omp end parallel
enddo

!! OpenACC for GPUs (without race condition):
!! loop over the groups
Nfac1 = Njfac
do ipass = 1, Npass_ift
   Nfac0 = Nfac1 + 1
   Nfac1 = fpass_ift(ipass)

!! loop over the edges
!$acc parallel
!$acc do
   do ifa = Nfac0, Nfac1
      !! left element
      iel = intfac(1,ifa)

      !! right element
      ier = intfac(2,ifa)

      !! loop over Gauss quadrature points
      do ig = 1, Ngp
         !! contribution to the left element
         rhsel(*,*,iel) = rhsel(*,*,iel) - flux

         !! contribution to the right element
         rhsel(*,*,ier) = rhsel(*,*,ier) + flux
      enddo
   enddo
!$acc end parallel
enddo
```
6.3 Gauss-Jordan Elimination based Matrix Inversion

With the above mentioned face coloring scheme, one can port the explicit time marching loop onto GPU platforms pretty straightforward using OpenACC directives. However, it is generally difficult for the implicit algorithm to work efficiently on GPU platform. As mentioned earlier, while outperforming in terms of numbers compared with a typical CPU core, those on GPU card would have much less computation power and local cache memory. As a result, for optimal performance on GPU, the algorithm should be divided into smaller units, thus occupying more cores with the same amount of work. In addition, the amount of data each core processes should be kept as small as possible. The latter aspect is particularly important for solving a block-sparse system, which consists of a large amount of square sub-matrices of an identical size. The size of sub-matrices would cause the memory-bounded issue, especially for higher order method. For any implicit algorithm, one would at least need to perform a matrix inversion for the diagonal matrix $D$. In 3D Navier-Stokes solver, for either $DG(P_1)$ method or $rDG(P_1P_2)$ scheme, the size of the diagonal matrix would be $(20 \times 20 \times Nelem)$. Therefore, if the operation is mapped to one GPU core, the algorithm easily becomes heavily memory-bounded, especially when the dimension of the sub-matrix is large, resulting in serious performance penalty. In other words, the key to achieve higher speed up factor on GPU platform is fine granularity.

Usually, for computing the inverse of a given matrix, one would need to adopt either direct method or iterative scheme. As a matter of fact, GPU would prefer the iterative scheme for its simplicity and not requiring the extra storage. However, for inverting of the diagonal matrix $D$ in a unstructured 3D Navier-Stokes solver, one can only use direct method since the iterative methods are not able to invert the non-diagonal-dominant matrix, which is indeed the common case for unstructured grid. In our study, we would adopt Gauss-Jordan elimination (GJE) based direct inversion algorithm [102], whose sequential algorithm is given in Table 6.3. Since this method is highly compact and does not require any extra storage, it would be particularly efficient on GPUs. The outer $r$-loop corresponds to the index of the row being transformed in GJE. Pivoting is generally not required, but it can be included if necessary. Although GJE is not the fastest method to compute matrix inversion, it is rather parallelizable. Also, it can be performed in-place, avoiding any thread-private arrays. Both factors are highly beneficial for GPU implementations.

The algorithm must be written as three kernels due to the data dependencies, and can be found in Table 6.4. These three kernels have different mappings between GPU threads and loop iterations. Conditions have been translated into index manipulations, avoiding idle threads for skipped iterations and thus for better performance on the GPU platforms. Again, since the
Table 6.3: Gauss-Jordan elimination without pivoting for an $n \times n$ matrix.

```fortran
DO r = 1, n
  !
  DO u = 1, n
    !
    IF (u .NE. r) THEN
      !
      d(u,r) = -d(u,r) / d(r,r)
      d(r,u) = d(r,u) / d(r,r)
    !
    END IF
    !
  END DO
  !
  DO v = 1, n
    !
    IF ((u .NE. r) .AND. (v .NE. r)) THEN
      !
      d(u,v) = d(u,v) + d(u,r) * d(r,v) * d(r,r)
    !
    END IF
    !
  END DO
  !
END DO

! d(r,r) = 1 / d(r,r)
!
END DO
```

$r$-loop runs on a CPU, there are $3n$ kernels launches to complete the inversion. Also, this algorithm can be easily extend to arbitrary high-order schemes.
Table 6.4: ACC version of GJE without pivoting for $Nelem$ matrices of size $n \times n$

```fortran
DO r = 1, n
!$acc kernels
!$ kernel # 1
!$acc loop independent
  DO ip = 0, 2 * (n - 1) * nelem -1
    !
    ie = ip / 2 / (n - 1)
    a = ip - ie * 2 * (n - 1) + 1
    ie = ie + 1
    a = a * (a >= r)
    a = a + (a >= a + n)
    t = (a > r ? 1:0)
    a = a - t * n
    b = t * (a - r)
    u = r + b
    v = a - b
    d(u,v,ie) = (1 - 2 * t) * d(u,v,ie) / d(r,r,ie)
  ! END DO
! kernel # 2
!$acc loop independent
  DO ip = 0, (n - 1) * (n - 1) * nelem -1
    !
    ie = ip / (n - 1) / (n - 1)
    t = ip - ie * (n - 1) * (n - 1)
    u = t / (n - 1)
    v = t - a * (n - 1) + 1
    ie = ie + 1
    a = a + 1
    u = u + (u >= r)
    v = v + (v >= r)
    d(u,v,ie) = d(u,v,ie) + d(u,r,ie) * d(r,v,ie) * d(r,r,ie)
  ! END DO
! kernel # 3
!$acc loop independent
  DO ie = 1, nelem
    !
    d(r,r,ie) = 1 / d(r,r,ie)
  ! END DO
!$acc end kernels
END DO
```

99
6.4 Parallelization for Linear Solver

For the implicit temporal algorithm, one would need to solve a linear solver at each time step/stage. Using an elemental based data structure, the left-hand-side matrix is stored in upper, lower, and diagonal forms. The most widely used methods to solve the linear system are iterative solution methods and approximate factorization methods.

In our framework, a linear system of equations can be solved iteratively by using either the Symmetric Gauss-Seidel (SGS) method, or the Generalized Minimal Residual (GMRES) method developed by Saad and Schulz [135] with LU-SGS preconditioning [84,96], namely GMRES+LU-SGS. Although GMRES is among the most popular and efficient implicit algorithms in CFD [13, 14,15,27,35,45,57,84,87,89,90,91,96,132,152], it has some shortcomings for GPU accelerations. GMRES is one of the most robust Krylov methods, but it is also highly expensive in terms of storage, because all the computed vectors in the orthogonal sequence need to be retained. The GMRES algorithm requires one flux evaluation per time step and one flux evaluation per inner GMRES iteration. The primary storage is dictated by the LU-SGS preconditioning, which requires the upper, lower and diagonal matrices to be stored for every non-zero element in the approximate left-hand-side matrix. The additional storage associated to the GMRES algorithm for a 3D DG(P1)/rDG(P1P2) scheme for N-S solver is an array of size \((k + 2) \times Nelem \times 20\), where \(k\) is the number of search directions. These additional storage may not seem to be an issue when parallelization are performed on CPU platform, it would bring a bottleneck for the GPU computation due to the limited storage on each GPU device. Hence, we have not ported the GMRES solver onto the GPU platforms yet. However, note that for the CPU version of our RDGFLO solver, LUSGS+GMRES is the default solver for implicit schemes.

On the other hand, the Lower Upper-Symmetric Gauss-Seidel (LU-SGS) or SGS approximate factorization method is attractive due to its good stability properties and competitive computational cost. As a matter of fact, for structured grid, the LU-SGS or SGS solver on the GPU platform, it is very easy to vectorized for the sweeps would be performed by using hyperplanes \(i + j + k = const\), where \(i\), \(j\), and \(k\) are the indices of a grid cell. Thus, forward sweep updates point \((i, j, k)\) using already updated values at \((i - 1, j, k)\), \((i, j - 1, k)\), and \((i, j, k - 1)\), whereas backward sweep uses cells \((i + 1, j, k)\), \((i, j - 1, k)\) and \((i, j, k + 1)\). However, for the unstructured grid, one would need to reorder the element and generate the hyperplanes. Sharov and Nakahashi [137] has proposed an algorithm for doing so. Note that if this reordering algorithm is applied to the structured grid with taking the corner element \((1,1,1)\) as a starting element, one would have each hyperplanes as \(i + j + k = const\). Nevertheless, in our code, we simply reorder the elements using the greedy algorithm similar to the one we apply for the dual
face to eliminate the inherent data dependency. After renumbering, each groups/ hyperplanes should only include the elements that do not share a common face. In this case the lower matrix $L$ would be computed by the surrounded elements whose group marks are lower and the host element. Similarly, the upper matrix $U$ would be computed by the neighbor elements with higher group marks. Therefore, the sweeps are performed by increasing or decreasing the group mark numbers. And in each group the computation can be done concurrently, thus the algorithm can be vectorized. Also, similar to the face coloring method, this reordering is performed only once before the main time loop. The associated memory overhead of this method is minimal, since the only extra memory requirement is to store the group marks of each element, which make it suitable for GPU computing.

6.5 Multiple-GPU Computation

Nowadays, a top-of-the-line GPGPU card like the Nvidia Tesla K series can contain over two thousand stream processors, yet its memory size is relatively small in comparison with a regular CPU compute node. For example, a Tesla K20c Accelerator has 2496 stream processors and 5GB in memory, sufficient for scientific computing and business modeling in various disciplines. However, such a memory size is still far from enough even for some basic CFD simulation problems. In order to extend the memory, we can choose a more recent product like the Tesla K40 Accelerator (2880 stream processors, 12GB memory). But the pricing of such a new device is still prohibitive ($4440 before tax), which would make GPU computing a rather disadvantageous alternative to CPU computing in terms of cost-effectiveness. Instead, the use of a cluster of legacy GPGPU cards appears a more economical and practical choice for large-scale CFD computing. Unlike the case of single-GPU computing that requires no communication between the devices within the main loop of time steps, the major difference in designing a multi-GPU parallel model lies in the fact that the inter-GPU communication needs to be taken into account. In the present work, we adopt a parallelism scheme based on a message passing interface (MPI) programming paradigm, where the METIS library is used for the partitioning of a grid into subdomain grids of approximately the same size. Consequently, the data exchange is required based on the MPI library, and the workflow is sketched in Table 6.6, in which the <ACC> tag denotes an OpenACC acceleration-enabled region, and the <MPI> tag means that MPI routine calls are needed in the case that multiple CPUs or GPUs are invoked. Compared with the standard DG method, two extra MPI routine calls are required for the rDG method in parallel mode, due to the fact that the solution vector at the partition ghost elements need to be updated after each reconstruction scheme. Note that here the workflow is based on explicit time
marching method. However, the multi-GPU computation capabilities of $p$-multigrid method and implicit scheme are also enabled in similar manner.

Table 6.5: Workflow for the main loop over the explicit time iterations.

```
!! loop over time steps
DO itime = 1, ntime
   <ACC> Predict time-step size

   !! loop over TVDRK stages
   DO istage = 1, nstage
      !! P1P2 least-squares reconstruction
      <ACC> IF(nreco > 0) CALL reconstruction_ls(...)

      !! data exchange for partition ghost cells
      <MPI> IF(nprcs > 1) CALL exchange(...)  

      !! r.h.s. residual from diffusion
      <ACC> IF(nvisc > 0) CALL getrhs_diffusion(...)

      !! WENO reconstruction
      <ACC> IF(nreco > 0) CALL reconstruction_weno(...)

      !! data exchange for partition ghost cells
      <MPI> IF(nprcs > 1) CALL exchange(...)  

      !! r.h.s. residual from convection
      <ACC> CALL getrhs_convection(...)  

      !! update solution vector
      <ACC> CALL tvdrk(...)  

      !! data exchange for partition ghost cells
      <MPI> IF(nprcs > 1) CALL exchange(...)  
   ENDDO
ENDDO
```

Usually, grid partitioning is the first task we should carry out before the parallel code can read in a series of partitioned grids. The original grid generated by some grid generation software is called the global domain in our context. The global domain is then decomposed into local domains (subdomains) and each processor is allocated the workload for at least one local domains. Domain decomposition into a user-specified number of local domains of high quality is a very complex task. Thus the RDGFLO code employs the open-source software package METIS [64] for domain decomposition in the pre-processing stage. METIS is a set of
Table 6.6: Workflow for the data exchange and synchronization procedure.

```fortran
!! data exchange for partition ghost elements
SUBROUTINE exchange(...)

!! copy partition ghost elements from GPU to CPU
!$acc update host( unkno(:, :, nelem+1:nelem+1:naadje) )

!! initialize receive
DO i = 1,Ncomm
  s = exprc(i)
  dtype = irecv(i)
  CALL mpi_irecv(...)
ENDDO

!! send out partition-adjacent elements
DO i = 1,Ncomm
  r = exprc(i)
  dtype = isend(i)
  call mpi_isend(...)
ENDDO

!! wait for all sends and receives to complete
CALL mpi_waitall(...)

!! copy partition ghost elements from CPU to GPU
!$acc update device( unkno(:, :, Nelem+1:Nelem+1:Naadje) )

END SUBROUTINE exchange
```

serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes developed in the Krypis Lab at the Department of Computer Science and Engineering at the University of Minnesota in the Twin Cities of Minneapolis. Here, one example is shown in Fig. 6.1 with a tetrahedral grid with 253,577 elements for flow over a Boeing 747 aircraft.

The data structures for parallel communication between different threads are organized in three categories: (1) elemental geometric information, (2) elemental solution vector and (3) statistical indicators. At the beginning of a parallel computation, each local domain ASCII file is read in by one thread and thanks to METIS, the total element number per thread is as equal as possible. This distribution is maintained throughout the whole computation with a static load balancing. In order to implement the first two categories, we need to create the so-called partition boundary ghost elements for the boundary faces of the local domains, which physically connect to other local domains if mapped into the graph of the global domain. The partition boundary ghost elements for a particular local domain resemble the ones from its partitioning
Figure 6.1: Domain decomposition by METIS for a tetrahedral grid (only surface shown) for flow past a Boeing 747 aircraft, 16 partitions.
adjacent local domains as they should have the same geometric information like the elemental mass matrix coefficients, coordinates of the cell centers, and normalization factors ($\Delta x$, $\Delta y$, $\Delta z$), and the same solution vectors during the advancement in time. The geometric information for the partition boundary ghost elements is exchanged at initialization of the computation and does not need synchronization since the grid is kept static under the current development. The statistical indicators mean the global residual vectors, global $L^2$ norm and some other suggestive indices dumped every user-specified number of time steps. These indicative figures are computed locally and finally collected and reported by one thread (usually Rank 0). An exchange object is built which possesses a send and receive buffer that contain the data to be exchanged between threads. The buffers are the consecutive partition boundary face-based degrees of freedom which are projected from the elemental degrees of freedom. Also keep in mind that the relative orientation of the corresponding partition boundary faces should be correct.

The communication in RDGFLO in parallel mode is managed by the described exchange object above, which utilizes nothing but the necessary standard MPI commands like nonblocking send, nonblocking receive and wait commands. In the present work, the data exchange between different GPU devices is required based on the MPI library, and the workflow is sketched in Table 6.6, where the continuous memory corresponding to the solution vector of the partition ghost elements (from $(Nelem + 1)$-th element to $(Nadje)$-th element) is first copied from the GPUs to the host CPUs, and then copied back to the GPUs after a non-blocking communication procedure. As one has observed, the overall inter-GPU communication procedure has to go through the host CPUs. In fact, the so-called GPUDirect [136] techniques, which are intended to realize direct communication between the GPUs, has yet to mature. In addition, most of the recently reported GPUDirect solutions would more or less require some unique software customization. Furthermore, even if these techniques are achievable within a durable effort on our current HPC resource, the resulting code is unlikely to be portable across the platforms. Therefore those are the main reasons that we do not anticipate the GPUDirect techniques to be incorporated into the framework of our OpenACC-based GPU code, unless more fundamental improvement in terms of hardware structures and OpenACC specifications would take place in the GPGPU industry.
Chapter 7

Numerical Results

The content of this chapter is split into numerical experiments of flow simulation (§7.1) and model equation for hyperbolic rDG methods in (§7.2). For all the GPU accelerated test cases, computation is carried out on an A Root Cluster for Research into Scalable Computer Systems (ARC) running the Red Hat 4.1.2-51 Linux operating system at the Department of Computer Science, North Carolina State University, with the Fortran 90 (International Business Machines Corporation) code compiled using the PGI Accelerator (The Portland Group, Inc., Lake Oswego, OR, USA) with OpenACC + OpenMPI package. As for the CPU only test cases in §7.2, simulation is deduced on a MacBook Pro (13-inch, Mid 2012) running the macOS High Sierra operating system (version 10.13.1) with 2.9 GHz intel cores i7 processor and 8GB 1600 MHz DDR3 memory. The code is written in Fortran 90 and compiled using GNU Fortran (gfortran) compiler.

To evaluate the speedup of GPU versus CPU, we compare the running time measured by using the GPU code on one Nvidia Tesla K20c card with that measured by using the equivalent CPU code on one compute node (16 CPU cores). Also, a verification test would be carried out with an absolute error tolerance of $1.0 \times 10^{-12}$ that compares the numerical results obtained by the GPU code with those by the equivalent CPU code. To assess the scalability of our code on multiple GPUs, we performed a weak scaling experiment, that assesses how the solution time varies with the number of GPU cards for a fixed problem size per GPU card, on up to eight Nvidia Tesla C2050 cards for each test case. Some parameters of these two GPU cards are shown in Table 7.1 and Fig. 7.1.
Table 7.1: Part of the Nvidia GPGPU resource on the NCSU’s ARC cluster (donated by NVIDIA).

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount</th>
<th>Stream processors</th>
<th>Memory</th>
<th>Bandwidth (GB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla C2050</td>
<td>9</td>
<td>448</td>
<td>3 GB</td>
<td>144</td>
</tr>
<tr>
<td>Tesla K20c</td>
<td>3</td>
<td>2496</td>
<td>5 GB</td>
<td>200</td>
</tr>
</tbody>
</table>

Figure 7.1: The NVIDIA Tesla GPGPUs used in the present work: (a) C2050 (448 stream processors, 3GB memory); (b) K20c (2496 stream processors, 5GB memory).

The unit time $T_{unit}$ is calculated as

$$T_{unit} = \frac{T_{wall-clock} \times Ngpus}{Ntime \times Nelem} \times 10^6 \text{ (microsecond)},$$  (7.1)

where $T_{wall-clock}$ refers to the wall-clock time recorded for completing the entire time marching loop with a given number of time steps $Ntime$ by using an amount of $Ngpus$ GPU cards, not including the start-up procedures, initial/end data translation, and solution file dumping. Note that there has not been a commonly accepted GPU benchmark testing for unstructured CFD solvers. Therefore, although speedup factors based on specific hardware models are not particularly preferable in a strict sense, yet it is not uncommon that they are used in literature [32, 82, 83, 155, 156].

7.1 Flow Simulation

In this section, a variety of numerical experiments of compressible inviscid and viscous flow problems computed by the rDG methods are presented. The first two cases are the inviscid flow
governed by Euler equations. The first one simulates the subsonic flow past a sphere to first verify the order of accuracy of the developed reconstructed Discontinuous Galerkin methods and then assess the performance of the developed methods on GPU devices. The second test case is to assess the performance of the rDG methods for a transonic flow over a complex geometrical configuration by simulating a transonic flow over a complete Boeing 747 aircraft. Next, three viscous test cases simulating the viscous flow by solving the Navier-Stokes equations on the GPU devices are carried out. The third case is to simulate a laminar flow past a sphere. The fourth test case is the quasi-2D lid-driven square cavity problem. This test case is carried out to verify the implementation of OpenACC for hexahedral elements and also to demonstrate the advantage of rDG(P₁P₂) scheme. It is also presented to assess and verify our rDG methods for computing complicated flows of practical importance. The final test case is the 3D Taylor-Green vortex flow problem to assess the accuracy and performance of rDG(P₁P₂) for the Direct Numerical Simulation (DNS).

### 7.1.1 Subsonic Flow past a Sphere

In this numerical example, an inviscid subsonic flow past a sphere at a freestream Mach number of \( M_\infty = 0.5 \) is chosen to assess the performance of the DG(P₁) and rDG(P₁P₂) methods for external flows. First of all, we conduct a grid refinement study with a sequence of the three successively refined tetrahedral grids for computation are shown in Fig. 7.2 respectively: the coarse grid (Level 1) shown in Fig. 7.2a consists of 535 elements, 167 nodes and 244 boundary faces; the medium grid (Level 2) shown in Fig. 7.2b consists of 2,426 elements, 589 nodes and 640 boundary faces; and the fine grid (Level 3) shown in Fig. 7.2c consists of 16,467 elements, 3,425 nodes and 2,372 boundary faces. The cell size is halved between consecutive grids. Note that only a quarter of the configuration is modeled due to symmetry of the problem.

The following \( L^2 \) norm of the entropy production is used as the error measurement for this steady-state inviscid flow problems

\[
\|\varepsilon\|_{L^2(\Omega)} = \sqrt{\int_{\Omega} \varepsilon^2 \, d\Omega} = \sqrt{\sum_{i=1}^{N_{\text{elem}}} \int_{\Omega_i} \varepsilon^2 \, d\Omega},
\]

(7.2)

where the entropy production \( \varepsilon \) is defined as

\[
\varepsilon = \frac{S - S_\infty}{S_\infty} = \frac{p}{p_\infty} \left( \frac{\rho_\infty}{\rho} \right)^\gamma - 1.
\]

(7.3)

Note that the entropy production, where the entropy is defined as \( S = (p/\rho)^\gamma \), is a very good
criterion to measure accuracy of the numerical solutions, since the flow under consideration is isentropic.

The order of accuracy for spatial discretization of a scheme can be assessed in a piecewise manner between two successively refined grids with a characteristic grid size $h$ and $h/2$ respectively, and computed as such a slope: the logarithmic difference of two $L^2$ norms over the difference of their corresponding logarithmic cell sizes expressed in the following

$$m = \frac{\log_{10} \varepsilon_h - \log_{10} \frac{\varepsilon_h}{2}}{\log_{10} h - \log_{10} \frac{h}{2}} = \frac{\log_{10} \frac{\varepsilon_h}{\varepsilon_{h/2}}}{\log_{10} 2}, \quad (7.4)$$

where $m$ is taken from the denotation of the rDG($P_nP_m$) scheme with a designed $O(h^{n+1})$ order of accuracy. The actual order of accuracy for a specific problem can be obtained by computing on a series of globally successively refined grids and taking an arithmetic averaged value of these piecewise slopes. The DG($P_1$) and rDG($P_1P_2$) methods are therefore expected to deliver the $O(h^2)$ and $O(h^3)$ order of accuracy, respectively.

Furthermore, a definition of the characteristic grid size $h$ is required in order to plot a figure of piecewise slopes with respect $h$ for the purpose of demonstration, although it is not required to compute the slope itself, as one can see from the above equation. $h$ can be defined as the actual cell sizes of a series of globally successively grids. Alternatively, $h$ can be computed in terms of total number of degree of freedom for a grid as below

$$h = \frac{1}{Ndimm \sqrt[3]{Ndegr} \times Nelem}, \quad (7.5)$$

where the index of root $Ndimm$ is the dimension of a problem type: $Ndimm = 1$ for 1D-type problems where grids are refined in one direction only i.e., $x$-direction and $Nelem$ grows linearly; $Ndimm = 2$ for 2D-type problems where grids are refined in a plane i.e., $x$-$y$ plane and $Nelem$ grows quadratically; $Ndimm = 3$ for 3D-type problems where grids are refined in 3D space and $Nelem$ grows cubically.

Computations on GPUs are first conducted on the first three grids (Level 1, 2 and 3) by using the DG($P_1$) and rDG($P_1P_2$) methods to obtain a quantitative measurement of the discretization errors and order of convergence, as shown in Table 7.2. In average, both DG($P_1$) and rDG($P_1P_2$) have achieved almost a formal order of accuracy of convergence, being 2.00 and 3.01, respectively, convincingly demonstrating the benefits of using the reconstructed DG method. Fig. 7.3, Fig. 7.4 and Fig. 7.5 show the computed density, pressure and Mach number contours in the flow field obtained by DG($P_1$ and rDG($P_1P_2$) on the three grids, respectively.
One can see that the results obtained by rDG(P₁P₂) are more accurate than the ones obtained by DG(P₁). All the presented results here are verified by a handmade diff program with a user-defined absolute error tolerance of $1.0 \times 10^{-12}$, indicating that the GPU code and the CPU code produced the identical solution on each grid.

Table 7.2: Discretization errors and convergence rates obtained on the three successively refined tetrahedral grids for inviscid subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$.

<table>
<thead>
<tr>
<th>Grids</th>
<th>Elements</th>
<th>$L^2$ norm (P₁)</th>
<th>Order (P₁)</th>
<th>$L^2$ norm (P₁P₂)</th>
<th>Order (P₁P₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
<td>535</td>
<td>-0.1732E+01</td>
<td>–</td>
<td>-0.196E+01</td>
<td>–</td>
</tr>
<tr>
<td>Medium</td>
<td>2,426</td>
<td>-0.2302E+01</td>
<td>1.895</td>
<td>-0.284E+01</td>
<td>2.924</td>
</tr>
<tr>
<td>Fine</td>
<td>16,467</td>
<td>-0.2933E+01</td>
<td>2.094</td>
<td>-0.377E+01</td>
<td>3.094</td>
</tr>
</tbody>
</table>

Figure 7.2: A sequence of three successively globally refined tetrahedral grids for inviscid subsonic flow past a sphere.

Next, a strong scaling test is carried out for rDG(P₁P₂) with explicit TVDRK3 method on a sequence of four successively refined tetrahedral grids on a NVIDIA K20c GPU card. Similarly, the grids size are halved successively. The most coarser grid here has 2,426 tetrahedral elements (the Medium mesh on the verification test), while the finest mesh contains 966,497 cells. The
Figure 7.3: Computed density contours in the flow field by the DG(P_1) (a–c) and rDG(P_1P_2) (d–f) on the tetrahedral grids for inviscid subsonic flow past a sphere at $M_\infty = 0.5$. 
Figure 7.4: Computed pressure contours in the flow field by the DG(P₁) (a–c) and rDG(P₁P₂) (d–f) on the tetrahedral grids for inviscid subsonic flow past a sphere at $M_\infty = 0.5$. 

112
Figure 7.5: Computed Mach number contours in the flow field by the DG(P₁) (a–c) and rDG(P₁) (d–f) on the tetrahedral grids for inviscid subsonic flow past a sphere at $M_\infty = 0.5$. 
strong scaling results can be found in Table 7.3. As one has observed, the OpenACC GPU code does not gain much advantage for a small-scale problem like 2426 elements. With adequate grid size like 124,706 and 966,497 elements, the advantage of GPU is then fully demonstrated, as speedup factor of up to 22.6 has been achieved by comparing with the CPU code.

Table 7.3: Timing measurements of using TVDRK3+rDG(P₁P₂) methods for subsonic flow past a sphere.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>T_{\text{unit}} (ms)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,426</td>
<td>2.23</td>
<td>19.6</td>
<td>8.8</td>
<td></td>
</tr>
<tr>
<td>16,467</td>
<td>1.20</td>
<td>20.3</td>
<td>17.0</td>
<td></td>
</tr>
<tr>
<td>124,706</td>
<td>1.03</td>
<td>20.3</td>
<td>19.6</td>
<td></td>
</tr>
<tr>
<td>966,497</td>
<td>0.97</td>
<td>22.1</td>
<td>22.3</td>
<td></td>
</tr>
</tbody>
</table>

Next, a weak scaling test is carried out on a sequence of four successively refined tetrahedral grids, which contain approximately half million, one million, two million, and four million elements, respectively, as shown in Table 7.4. These four grids correspond to the use of one, two, four, and eight NVIDIA Tesla C2050 GPU cards respectively, ensuring an approximately fixed problem size per GPU card. The detailed timing measurements are presented in Table 7, showing the statistics of unit running time and parallel efficiency obtained on each grid. In addition, Fig. 7.6a and Fig. 7.6b display the variations of the unit running time and parallel efficiency with respect to the number of GPUs, indicating that both the DG(P₁) and rDG(P₁P₂) methods have achieved good parallel efficiency in the case of eight GPUs, being 87.7% and 88.2% respectively. The primary loss of efficiency in multi-GPU mode is due to the overheads in GPU-to-CPU and CPU-to-GPU data copies, and MPI communication and synchronization between the host CPUs. Above all, this parallel rDG solver based on the OpenACC directives exhibits a competitive scalability for computing inviscid flow problems on multiple GPUs.

Before we step into the fully implicit DG/rDG scheme on GPU platform, another strong scaling timing test is carried out on same sequence of four successively refined tetrahedral grids as the ones we ran the previous strong scaling with TVDRK3. The simulations are done by GPU-accelerated DG(P₁) or rDG(P₁P₂) with p-multigrid method and original CPU code, so that we can see the efficiency of the GPU computing. The detailed timing measurements are
Table 7.4: Timing measurements of weak scaling obtained on a cluster of NVIDIA Telsa C2050 GPU cards for using rDG methods with TVDRK3 for subsonic flow past a sphere.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Ngpu</th>
<th>(T_{\text{unit}}(\text{ms}))</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DG(P1)</td>
<td>rDG(P1P2)</td>
</tr>
<tr>
<td>501,972</td>
<td>1</td>
<td>0.57</td>
<td>1.57</td>
</tr>
<tr>
<td>1,015,570</td>
<td>2</td>
<td>0.60</td>
<td>1.61</td>
</tr>
<tr>
<td>1,999,386</td>
<td>4</td>
<td>0.63</td>
<td>1.69</td>
</tr>
<tr>
<td>4,029,430</td>
<td>8</td>
<td>0.65</td>
<td>1.78</td>
</tr>
</tbody>
</table>

Presented in Table 7.5, showing the statistics of unit running time. Note that the reason we do not present the data for the \(p\)-multigrid rDG(P1P2) for the most fine mesh is that it would exceed the K20c GPU card memory. From the results we can see that GPU performs better when it comes to larger \(Nelem\), and can achieve higher speedup factor when we use DG(P1).

To show the effect of the \(p\)-multigrid, we would show the convergence history of the next fine mesh \((Nelem = 124,706)\), thus we can include rDG(P1P2) with \(p\)-multigrid on GPU platform without the memory issue, which are shown on Fig. 7.7a and Fig. 7.7b. Clearly, the \(p\)-multigrid shows its effect on accelerating the convergence procedure. This case shows that the GPU accelerated \(p\)-multigrid DG method is effective and the speed up is promising.

Table 7.5: Timing measurements of using \(p\)-multigrid DG/rDG methods for subsonic flow past a sphere.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>(T_{\text{unit}}(\text{ms})) by (p)-multigrid DG(P1)</th>
<th>(T_{\text{unit}}(\text{ms})) by (p)-multigrid rDG(P1P2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td>2,426</td>
<td>7.83</td>
<td>10.19</td>
</tr>
<tr>
<td>16,467</td>
<td>3.22</td>
<td>12.13</td>
</tr>
<tr>
<td>124,706</td>
<td>1.00</td>
<td>13.36</td>
</tr>
<tr>
<td>966,497</td>
<td>0.67</td>
<td>15.97</td>
</tr>
</tbody>
</table>

A final strong scaling timing test is carried out on the same sequence of tetrahedral girds
Figure 7.6: Plot of the timing measurements for inviscid subsonic flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using TVDRK3+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.

Figure 7.7: Subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$: (a) Comparison of convergence history versus CPU time between TVDRK3 and $p$-multigrid methods for DG(P1); (b) Comparison of convergence history versus CPU time between TVDRK and $p$-multigrid methods for rDG(P1P2).
Table 7.6: Timing measurements of using BDF1+DG/rDG methods for subsonic flow past a sphere.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{\text{unit}}$ (ms) by BDF1+DG(P1)</th>
<th>$T_{\text{unit}}$ (ms) by BDF1+rDG(P1P2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td>2,426</td>
<td>41.63</td>
<td>74.60</td>
</tr>
<tr>
<td>16,467</td>
<td>15.18</td>
<td>77.43</td>
</tr>
<tr>
<td>124,706</td>
<td>11.18</td>
<td>80.99</td>
</tr>
</tbody>
</table>

to assess the performance of the implicit rDG method on GPU. The temporal discretization is BDF1 and SGS method are used as the linear solver in each time step. The simulation are performed again on a single NVIDIA Telsa K20c GPU card. The detailed timing measurements are presented in Table 7.6, showing the statistics of unit running time. Similarly, the GPU performs better when it comes to larger Nelem, and can higher speed up factor when the DG(P₁) is used. For better illustration, a breakdown of the averaged one main loop run times is given in Fig. 7.8. The result is based on running implicit rDG(P₁P₂) on the mesh with 124,706 tetrahedral elements. From the breakdown, we can see the primary bottleneck for the current GPU implementation is for the left hand side Jacobian computing, which takes large percentage of the time while the speed up factor is only 3.29. The reason behind this is again the memory bound issue, since we have large amounts of memory assessed of a relatively small amount of computation. This part of code is still implemented in a coarse-grained fashion, as the computation for each element is mapped to one GPU thread, making them highly memory-bound. To obtain full performance on GPUs, an effort needs to be taken to develop fine-grained versions of the related subroutines.

Figure 7.8: Averaged one main time loop run times (in seconds) for running implicit rDG(P₁P₂) on the finest mesh for the inviscid sphere case.
Lastly, a weak scaling test is carried out on another sequence of four successively refined tetrahedral grids. These four grids correspond to the use of one, two, four, and eight NVIDIA Tesla C2050 GPU cards respectively, ensuring an approximately fixed problem size per GPU card in each case. The timing results are shown in Table 7.7, indicating the unit running time and parallel efficiency obtained on each grid. Additionally, Fig. 7.9a and Fig. 7.9b display the variations of the unit running time and parallel efficiency with respect to the number of GPUs, indicating that both the DG(P1) and rDG(P1P2) methods have achieved good parallel efficiency in the case of eight GPUs, being 89.7% and 88.3% respectively. Both the DG(P1) and rDG(P1P2) methods have achieved good parallel efficiency in the case of multi-GPU computing.

Table 7.7: Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for using rDG methods with BDF1 for subsonic flow past a sphere.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>N gpu</th>
<th>$T_{\text{unit (ms)}}$</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DG(P1)</td>
<td>rDG(P1P2)</td>
</tr>
<tr>
<td>62,481</td>
<td>1</td>
<td>26.25</td>
<td>31.69</td>
</tr>
<tr>
<td>124,706</td>
<td>2</td>
<td>28.48</td>
<td>34.00</td>
</tr>
<tr>
<td>249,945</td>
<td>4</td>
<td>28.80</td>
<td>34.89</td>
</tr>
<tr>
<td>501,972</td>
<td>8</td>
<td>29.28</td>
<td>35.87</td>
</tr>
</tbody>
</table>
Figure 7.9: Plot of the timing measurements for inviscid subsonic flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using BDF1+DG/rDG schemes: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.

7.1.2 Transonic Flow over a Boeing 747 Aircraft

In the second case, we choose a transonic flow past a complete Boeing 747 aircraft at a freestream of Mach number of \( M_\infty = 0.85 \) and an angle of attack of \( \alpha = 2^\circ \). This case could test the ability of computing complex geometric configurations by a OpenACC-based reconstructed discontinuous Galerkin solver. The configuration of Boeing 747 includes the fuselage, wing, horizontal and vertical tails, under-wing pylons, and flow-through engine nacelle. The grid used in the computation is tetrahedron. Similarly, only half of the aircraft is modeled due to the symmetry of the problem. Computation is first conducted on a tetrahedral grid containing 253,577 elements, as shown in Fig. 7.10a. The computed steady-state density, Mach number and pressure contours obtained by rDG (P1P2) are illustrated in Fig. 7.10b, Fig. 7.10c and Fig. 7.10d respectively. One can observe that the shock waves on the upper surface of the wing are well captured, confirming the accuracy, robustness, and efficiency of our method for computing complicated flows of practical importance. A diff check with the absolute error tolerance of \( 1 \times 10^{-12} \) indicates that the GPU code produced the identical solution data to those by the equivalent CPU code.

Secondly, a scaling test for the K20c GPU card is conducted by running TVDRK3 with rDG(P1P2), with the timing measurements presented in Table 7.8. Speedup factor of up to 24.5 can be achieved for the GPU code by comparing with the CPU code running on one CPU.
Figure 7.10: Inviscid transonic flow over a Boeing 747 aircraft computed using rDG(P₁P₂), (a) the tetrahedral grid used for computation ($Nelem = 253, 577$, $Npoin = 48, 851$, $Nbfac = 23, 616$, ) (b) computed density contours, (c) computed Mach number contours, and (d) computed pressure contours.
compute node. Above all, the highest speedup factors observed in this test case are similar to those in the first test case, indicating the consistent and stable performance of the resulting OpenACC GPU code for computing various flow conditions and geometric configurations. Later, a corresponding weak scaling test is carried out for up to eight NVIDIA C2050 GPU cards is used by simulating the transonic flow over a Boeing 747 aircraft with the results shown in Table 7.9. The unit running time and the efficiency can also be found in Fig. 7.11a and Fig. 7.11b respectively. The results has shown that the good parallel efficiency of the presented scheme for the computation on the complex geometry.

Table 7.8: Timing measurements of using TVDRK3+rDG(P_{1}P_{2}) methods for inviscid transonic flow over a Boeing 747 aircraft \( M_{\infty} = 0.85, \alpha = 2^\circ \).

<table>
<thead>
<tr>
<th>Nelem</th>
<th>( T_{\text{unit}} ) (ms)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>253,577</td>
<td>1.27</td>
<td>27.10</td>
<td>21.2</td>
<td></td>
</tr>
<tr>
<td>1,025,170</td>
<td>1.17</td>
<td>27.70</td>
<td>24.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.9: Timing measurements of weak scaling obtained on a cluster of NVIDIA Telsa C2050 GPU cards for using rDG methods with TVDRK3 for transonic flow over a Boeing 747 aircraft.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Ngpu</th>
<th>( T_{\text{unit}} ) (ms)</th>
<th>Parallel efficiency</th>
<th>DG(P_{1})</th>
<th>rDG(P_{1}P_{2})</th>
<th>DG(P_{1})</th>
<th>rDG(P_{1}P_{2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>127,947</td>
<td>1</td>
<td>0.97</td>
<td>2.17</td>
<td>-</td>
<td>-</td>
<td>95.0%</td>
<td>97.3%</td>
</tr>
<tr>
<td>253,577</td>
<td>2</td>
<td>1.02</td>
<td>2.23</td>
<td>95.0%</td>
<td>97.3%</td>
<td>95.0%</td>
<td>97.3%</td>
</tr>
<tr>
<td>512,280</td>
<td>4</td>
<td>1.08</td>
<td>2.31</td>
<td>89.8%</td>
<td>93.9%</td>
<td>89.8%</td>
<td>93.9%</td>
</tr>
<tr>
<td>1,025,170</td>
<td>8</td>
<td>1.12</td>
<td>2.39</td>
<td>86.6%</td>
<td>90.8%</td>
<td>86.6%</td>
<td>90.8%</td>
</tr>
</tbody>
</table>

Next, another strong scaling test is carried out on the same grids, which can be found in Table 7.10. Clearly, we can see that the GPU achieves higher speed up factor with \( p \)-multigrid DG(P_{1}). Again, we do not have the data for \( p \)-multigrid rDG(P_{1}P_{2}) for the fine mesh due the
Figure 7.11: Plot of the timing measurements for inviscid transonic flow over a Boeing 747 aircraft with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using TVDRK3+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.

memory limitation of the available GPU cards. Fig. 7.12a and Fig. 7.12b display a comparison of convergence histories versus CPU time between TVDRK3 DG/rDG on CPU and $p$-multigrid DG/rDG methods on GPU platform. The $p$-multigrid method is much faster than its explicit TVDRK counterpart for this test case. The excellent acceleration of the $p$-multigrid method is again demonstrated for this flow problem.

Table 7.10: Timing measurements of using $p$-multigrid DG methods for transonic flow past a Boeing 747 aircraft.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{\text{unit}}(\text{ms})$ by $p$-multigrid DG($P_1$)</th>
<th>$T_{\text{unit}}(\text{ms})$ by $p$-multigrid rDG($P_1P_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td>253,577</td>
<td>1.09</td>
<td>13.43</td>
</tr>
<tr>
<td>1,025,170</td>
<td>0.81</td>
<td>17.19</td>
</tr>
</tbody>
</table>

As for implicit rDG scheme on the GPU cluster, both strong scaling test on NVIDIA K20c GPU and weak scaling test on up to 8 NVIDIA C2050 GPUs are carried out. The timing results
Figure 7.12: Transonic flow past a Boeing 747 aircraft at a free-stream Mach number of $M_\infty = 0.85$ and a angle of attack of $\alpha = 2^\circ$: (a) Comparison of convergence history versus CPU time between TVDRK and $p$-multigrid methods for DG(P$_1$); (b) Comparison of convergence history versus CPU time between TVDRK and $p$-multigrid methods for rDG(P$_1$P$_2$).

for the strong scaling can be found in Table 7.11. Speedup factor of up to 8.23 can be achieved for the GPU code by comparing with the CPU code running sequentially. The speed up factor is similar to the previous case, indicating that our method could handle complex geometry pretty well on GPU platform.

Table 7.11: Timing measurements of using BDF1+DG/rDG methods for transonic flow past a Boeing 747 aircraft.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{\text{init}}$ (ms) by BDF1+ DG(P$_1$)</th>
<th>$T_{\text{init}}$ (ms) by BDF1 +rDG(P$_1$P$_2$)</th>
<th>Speedup</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU</td>
<td>CPU</td>
<td>Speedup</td>
<td>GPU</td>
</tr>
<tr>
<td>64,066</td>
<td>11.86</td>
<td>90.84</td>
<td>7.66</td>
<td>15.30</td>
</tr>
<tr>
<td>127,947</td>
<td>11.02</td>
<td>90.66</td>
<td>8.23</td>
<td>13.47</td>
</tr>
</tbody>
</table>

As for the weak scaling test for implicit rDG shames, it is performed on a sequence of four successively refined tetrahedral grids. The timing measurement results are shown in Table 7.12. As one can see, each GPU card would have an approximately fixed problem size. Meanwhile,
Fig. 7.13a and Fig. 7.13b display the variations of the unit running time and parallel efficiency with respect to the number of GPUs, indicating that both the DG(P₁) and rDG(P₁P₂) methods have achieved good parallel efficiency in the case of eight GPUs, being 90.2% and 90.4% respectively. Therefore, the shown results indicate that the presented method could remain high parallel efficiency in this complex geometry case on multiple GPUs.

Table 7.12: Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for using rDG methods with BDF1 for transonic flow over a Boeing 747 aircraft.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Ngpu</th>
<th>$T_{\text{unit}}$(ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>DG(P₁)</td>
</tr>
<tr>
<td>64,066</td>
<td>1</td>
<td>25.58</td>
<td>-</td>
</tr>
<tr>
<td>127,947</td>
<td>2</td>
<td>26.57</td>
<td>96.3%</td>
</tr>
<tr>
<td>253,577</td>
<td>4</td>
<td>28.08</td>
<td>91.1%</td>
</tr>
<tr>
<td>512,280</td>
<td>8</td>
<td>28.35</td>
<td>90.2%</td>
</tr>
</tbody>
</table>
Figure 7.13: Plot of the timing measurements for transonic flow over a Boeing 747 aircraft with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using BDF1+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.
7.1.3 Laminar Flow past a Sphere

A laminar flow past a sphere at a freestream Mach number of $M_\infty = 0.5$ and Reynolds number of $Re = 118$ based on the diameter of the sphere is considered in this test case. The objective for this numerical example is to assess and verify the performance of the rDG methods for low Reynolds number flow problems. The initial condition is a uniform freestream with non-slip boundary conditions on the solid wall. Firstly, in order to verify the implementation of OpenACC parallel scheme for the viscous problem, computation is conducted on a coarse grid consisting of 119,390 tetrahedral elements, 22,530 grid nodes and 4,511 boundary nodes as shown in Fig. 7.14a. One can observe the coarseness of the triangular surface meshes near the sphere wall region. Note that only half of the configuration is modeled due to the symmetry of the problem. Fig. 7.14b displays the computed steady-state velocity streamtraces on the symmetry plane obtained by rDG ($P_1P_2$). As one can observe, the two trailing vortices are visually identical and symmetric to the center line. A diff check with an absolute error tolerance of $1 \times 10^{-12}$ indicates that the GPU code and the CPU code produced the identical solution data. In addition, the computed pressure contours and Mach number contours on the surface meshes in the flow field are displayed in Fig. 7.15a and Fig. 7.15b, respectively.

![Figure 7.14: (a) Plot of the surface meshes of the tetrahedral grid for laminar flow past a sphere at $M_\infty = 0.5$ and $Re = 118$; (b) plot of the computed velocity streamtraces on the symmetry plane using rDG($P_1P_2$).](image)
Secondly, a strong scaling test is conducted with the timing measurements presented in Table 7.13. One can obtain a speed factor of up to 18.5 comparing with one CPU processor when conducting the computation on a single NVIDIA K20c GPU card. It is interesting to find that the speedup factors obtained for the Navier-Stokes equations are lower than those for the Euler equations in the previous case, although the computational intensity in this case is obviously higher. In fact, the major latency is due to the code structure that the viscous and inviscid flux calculations are divided into two separate procedures, because the WENO reconstructed quadratic polynomials are only needed for computing the inviscid residuals. Therefore, the overheads in acceleration kernels within the r.h.s. computation are doubled. Indeed, the code could render higher efficiency if we chose to cluster the least-squares reconstruction and WENO reconstruction at the head of the r.h.s. process and merge the viscous and inviscid flux calculations into one face integral and one volumetric integral. However, study shows that the solution accuracy would be affected if we did so.

Thirdly a weak scaling test is carried out on a sequence of four successively refined tetrahedral grids, which contain approximately half million, one million, two million, and four million elements, respectively. These four grids correspond to the use of one, two, four, and eight Nvidia Tesla C2050 GPU cards respectively, ensuring an approximately fixed problem size per GPU.
Table 7.13: Timing measurements of using TVDRK3+\text{rDG}(P_1P_2)\) methods for laminar flow past a sphere.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{\text{unit}}$ (ms) by TVDRK3 + rDG(P_1P_2)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>200,416</td>
<td>1.63</td>
<td>28.87</td>
<td>17.8</td>
<td></td>
</tr>
<tr>
<td>925,925</td>
<td>1.53</td>
<td>28.57</td>
<td>18.5</td>
<td></td>
</tr>
</tbody>
</table>

card. The total number of time iterations is set to be 10,000 for all of these four grids, with the detailed timing measurements presented in Table 7.14. Fig. 7.16a and Table 7.16b display the variations of the unit running time and parallel efficiency with respect to the number of GPUs respectively. The parallel efficiency ratios of 90.4\% and 87.3\% were obtained for DG(P_1) and rDG(P_1P_2) using TVDRK3 in the case of eight GPUs, respectively, demonstrating the good scalability of the developed rDG solver on multiple GPUs for computing viscous flow problems on tetrahedral grids.

Table 7.14: Timing measurements of weak scaling obtained on a cluster of NVIDIA Telsa C2050 GPU cards for using rDG methods with TVDRK3 for laminar flow past a sphere.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Ngpu</th>
<th>$T_{\text{unit}}$ (ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>500,095</td>
<td>1</td>
<td>0.66</td>
<td>-</td>
</tr>
<tr>
<td>1,000,103</td>
<td>2</td>
<td>0.67</td>
<td>98.5% 97.7%</td>
</tr>
<tr>
<td>2,000,051</td>
<td>4</td>
<td>0.68</td>
<td>97.1% 93.6%</td>
</tr>
<tr>
<td>4,000,227</td>
<td>8</td>
<td>0.73</td>
<td>90.4% 87.3%</td>
</tr>
</tbody>
</table>
Figure 7.16: Plot of the timing measurements for laminar flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (NVIDIA Tesla C2050) using TVDRK3+DG/rDG scheme: (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.
7.1.4 Quasi-2D Lid Driven Square Cavity

A quasi-2D lid driven square cavity laminar flow at a Reynolds numbers of Re = 100, 1000, and 10,000 is considered in this numerical experiment. The cavity dimensions are 1 unit in the x and y directions, and 0.1 unit in the z direction. First the computation is conducted on four Nvidia Tesla C2050 GPU cards to verify if the GPU code can deliver the identical numerical solutions to those by the equivalent CPU code. A sparse hexahedral grid which was used in Reference [153], is used in the verification test as shown in Fig. 7.17a. This grid consists of $32 \times 32 \times 2$ grid points, which has only one element in the span-wise z direction. The grid points are clustered near the walls in the x and y directions, and the grid spacing is geometrically stretched away from the wall with the minimum value $h_{\min} = 0.005$ (equivalent to $y^+ = 3.535$). On the bottom and side walls, the no-slip, adiabatic boundary conditions are prescribed. Along the top “lid”, the no-slip, adiabatic boundary conditions along with a lid velocity $\mathbf{v}_B = (0.2, 0, 0)$ are prescribed. On the front and back walls, a symmetric boundary condition is prescribed. The computation is started with the velocity field at rest without perturbation, and terminated at a sufficiently large total number of time iterations, ensuring that a steady-state flow field is reached. The computed velocity streamtraces obtained by rDG($P_1P_2$) on the GPU code is displayed in Fig. 7.17b, Fig. 7.17c, and Fig. 7.17d, demonstrating the ability of the rDG($P_1P_2$) method to accurately resolve all the major vortices on this sparse grid. Meanwhile, Fig. 7.18, Fig. 7.19, and Fig. 7.20 displays the profiles of the computed normalized velocity components $u/u_B$ and $v/u_B$ by DG($P_1$) and rDG($P_1P_2$) that are plotted along the y and x center-lines respectively. The profiles by a second-order compressible finite volume solver based on a WENO reconstruction [151], namely rDG($P_0P_1$) in our rDG($P_nP_m$) framework, is also presented. As one can observe, only the profiles by rDG($P_1P_2$) matched well with the classical reference data by Ghia et al. [48], clearly demonstrating the superior accuracy of the rDG($P_1P_2$) method in the case of high Reynolds numbers and very sparse grid resolution. Above all, a diff check with an absolute error tolerance of $1.0 \times 10^{-12}$ indicates that the GPU code and the CPU code produced the identical solution data in this verification test.

Secondly, a strong scaling test is designed and conducted on two hexahedral grids, which contain 500,000 and 1,000,000 elements, respectively at Reynolds number of 10,000. The timing measurements are presented in Table 7.15. Speedup factor of up to 18.92 is achieved by comparing the unit running time obtained on the K20c GPU with those by the one CPU processor. Overall, we can see that the developed OpenACC GPU code renders consistent performance on different types of elements and a wide range of Reynolds number.

Lastly a weak scaling test is carried out on a sequence of four successively refined hexahedral
Figure 7.17: A quasi-2D lid driven square cavity flow at a lid velocity of $v_B = (0.2, 0, 0)$ and a series of Reynolds number: (a) The sparse hexahedral grid ($32 \times 32 \times 2$ grid points) used in the verification test, (b) $Re = 100$, (c) $Re = 1000$, and (d) $Re = 10000$. 
Figure 7.18: Profiles of the normalized velocity components $u/u_B$ and $v/u_B$ on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid driven square cavity at a lid velocity of $v_B = (0.2, 0, 0)$, and a Reynolds number of $Re = 100$.

Table 7.15: Timing measurements of using TVDRK3+rDG(P₁P₂) methods for quasi-2D lid driven square cavity problem at $Re = 10,000$.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{unit}$ by TVDRK3 + rDG(P₁P₂)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>253,577</td>
<td>1.97</td>
<td>36.57</td>
<td>18.56</td>
<td></td>
</tr>
<tr>
<td>1,025,170</td>
<td>1.93</td>
<td>36.53</td>
<td>18.92</td>
<td></td>
</tr>
</tbody>
</table>
Figure 7.19: Profiles of the normalized velocity components $u/u_B$ and $v/u_B$ on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid driven square cavity at a lid velocity of $v_B = (0.2, 0, 0)$, and a Reynolds number of $Re = 1000$. 
Figure 7.20: Profiles of the normalized velocity components $u/u_B$ and $v/u_B$ on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid driven square cavity at a lid velocity of $v_B = (0.2, 0, 0)$, and a Reynolds number of $Re = 10,000$. 
grids, which contain half million (1000 × 500), one million (1000 × 1000), two million (1000 × 2000), and four million (1000 × 4000) elements, respectively. These four grids correspond to the use of one, two, four, and eight NVIDIA Tesla C2050 GPU cards respectively, ensuring a fixed problem size per GPU card. The Reynolds number is set to be 10,000 for all of these four grids. The detailed timing measurements are presented in Table 7.16, showing the statistics of unit running time and parallel efficiency obtained on each grid. In addition, Fig. 7.21a and Fig. 7.21b displayed the variations of the unit running time and parallel efficiency with respect to the number of GPUs, demonstrating that both the DG(P_1) and rDG(P_1P_2) methods have achieved excellent parallel efficiency in the case of eight GPUs, being 96.3% and 97.2% respectively. Such a high scalability as obtained in this test case is mainly attributed to the use of hexahedral grids, which renders better load-balanced grid partitions and renumbered face groups than tetrahedral grids. Above all, this parallel rDG solver has exhibited strong scalability on multiple GPUs for computing viscous flow problems on hexahedral grids.

Table 7.16: Timing measurements of weak scaling obtained on a cluster of Nvidia Tesla C2050 GPU cards for a quasi-2D lid driven square cavity.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Ngpu</th>
<th>T_{unit}(ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DG(P_1)</td>
<td>rDG(P_1P_2)</td>
</tr>
<tr>
<td>500,000</td>
<td>1</td>
<td>1.04</td>
<td>3.53</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2</td>
<td>1.06</td>
<td>3.57</td>
</tr>
<tr>
<td>2,000,000</td>
<td>4</td>
<td>1.07</td>
<td>3.60</td>
</tr>
<tr>
<td>4,000,000</td>
<td>8</td>
<td>1.08</td>
<td>3.63</td>
</tr>
</tbody>
</table>
Figure 7.21: Plot of the timing measurements for a quasi-2D lid driven square cavity with a fixed problem size (approximately half million elements) per GPU card (Nvidia Tesla C2050): (a) Unit running time versus the number of GPU cards, (b) Parallel efficiency versus the number of GPU cards.
7.1.5 3D Taylor-Green Vortex

The Taylor-Green vortex flow problem, one of the benchmark cases in the 3rd International Workshop on high order CFD methods, is chosen in this test case to assess the accuracy and performance of DG(P1) and rDG(P1P2) for the Direct Numerical Simulation (DNS) of turbulent flows on GPU platform. This problem was originally designed to numerically study the dynamics of turbulence. The initial conditions are smooth, but the flow quickly transits to turbulence with the creation of small scales and begins to decay, mimicking homogeneous non-isotropic turbulence. The initial conditions are given by

\[
\begin{align*}
    u &= V_0 \sin \left(\frac{x}{L}\right) \cos \left(\frac{y}{L}\right) \cos \left(\frac{z}{L}\right), \\
    v &= -V_0 \cos \left(\frac{x}{L}\right) \sin \left(\frac{y}{L}\right) \cos \left(\frac{z}{L}\right), \\
    w &= 0, \\
    p &= p_0 + p_0 V_0^2 \left[ \frac{1}{16} \left( \cos \left(\frac{2x}{L}\right) + \cos \left(\frac{2y}{L}\right) \right) \left( \cos \left(\frac{2z}{L}\right) + 2 \right) \right],
\end{align*}
\]  

(7.6)

where \( p_0 = 1 \) and \( p_0 = 1/\gamma \).

The flow is initialized to be isothermal \( (p/\rho = p_0/\rho_0 = RT_0) \). To minimize the effects of compressibility, the free-stream Mach number is set to 0.1. The Reynolds number in this case is 1,600, which corresponds to a peak Taylor microscale Reynolds number of about 22. The flow is computed in a periodic and square box, which spans \([-\pi L, \pi L]\) in each coordinate direction. The physical duration of the computation is 20 based on the characteristic convective time defined as \( t_c = L/V_0 \), i.e., \( t_{final} = 20t_c \). Before we look into this case for details, some useful concepts should be introduced in advance. The temporal evolution of the kinetic energy integrated on the domain \( \Omega \) is defined as

\[
E_k = \frac{1}{\rho_0 \Omega} \int \rho \frac{\mathbf{v} \cdot \mathbf{v}}{2} d\Omega, \quad (7.7)
\]

The temporal evolution of the kinetic energy dissipation rate is given by,

\[
e = e_1 + e_2 + e_3 = -\frac{dE_k}{dt}, \quad (7.8)
\]

where

\[
\begin{align*}
    e_1 &= \frac{1}{\Omega} \int 2\mu s_{ij} s_{ij} d\Omega, \\
    e_2 &= \frac{1}{\Omega} \int 2\mu_{\nu} u_{kk} u_{kk} d\Omega, \\
    e_3 &= -\frac{1}{\Omega} \int p u_{kk} d\Omega,
\end{align*}
\]  

(7.9)
and
\[ s_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right), \tag{7.10} \]
is the strain-rate tensor.

The temporal evolution of the enstrophy integrated on the domain \( \Omega \) is defined as
\[ \varepsilon = \frac{1}{\rho_0 \Omega} \int \frac{\omega \cdot \omega}{2} d\Omega, \tag{7.11} \]

In this case, the gas is assumed to have zero bulk viscosity, i.e., \( \mu_v = 0 \). Therefore, the dissipation due to the bulk viscosity is always equal to zero, meaning that \( e_2 = 0 \). As a matter of fact, these three contribution terms to the kinetic energy dissipation rate could also be used to assess the accuracy of the numerical solutions. Since the flow is nearly incompressible, the dissipation due to the pressure-dilatation term \( e_3 \) can be expected to be small. Hence, \( e \) could be approximated using \( e_1 \). Time histories of the computed \( e, e_1, \) and \( e_3 \) can be used to test the accuracy of the presented scheme. Note that \( e_3 \) has a significant bias, contributing some net positive kinetic energy dissipation. With increasing mesh refinement, the biased pressure-dilatation term should decrease toward zero.

First of all, a verification test is carried out using rDG(P1P2) method on 4 hexahedral meshes, which have \( 40^3, 81^3, 161^3, \) and \( 256^3 \) elements respectively. Note that the RK4 explicit time stepping method is applied in all the computation in this unsteady case. Fig. 7.22 shows the computed vortex detection criterion \( Q_2 \) at \( t = 8t_c \), on the finest mesh. One can observe that the vortex structure obtained by the rDG(P1P2) solution is very similar to Bull’s work \([23]\).

Fig. 7.23, Fig. 7.24 and Fig. 7.25 compare the time history of the kinetic energy \( E_k \), the kinetic energy dissipation rate \( e \), and the enstrophy computed from the data at the space-time quadrature points, respectively, for all four grids, with the result from an incompressible simulation using a spectral code on a mesh of \( 512^3 \) grid points \([145]\). There are significant numerical dissipation on the coarse mesh and the point of the peak dissipation is poorly captured. The higher resolution mesh would yield to more accurate solution. Clearly, the results from rDG(P1P2) agree well with the reference solution. Again, a diff check with an absolute error tolerance of \( 1.0 \times 10^{-12} \) indicates that the GPU code can reproduce the results of the CPU version.

Secondly, a strong scaling test is carried out to test the performance of the rDG(P1P2) method on a NVIDIA K20c GPU card. Due to the limitation of the GPU card, only the grids consist of \( 40^3 \) and \( 81^3 \) elements are used in this strong scaling test. The timing measurement results are shown in Table 7.17. Though it seems that the maximum speed up factor is less
than the previous case, one should be aware that a relatively finer mesh could be applied in the test to access better performance of the GPU card. Nevertheless, for a DNS type problem, the parallel efficiency of multi-GPU computing would be more instrested.

Therefore, a following weak scaling test is carried out for a set of four hexahedral grids, which consists around half million, one million, two million and four millions respectively. Again, the computation would be carried out on up to eight NVIDIA C2050 GPU devices. The detailed timing measurements are shown in Table 7.18. Additionally, Fig. 7.26a and Fig. 7.26b are used to display the variations of the unit running time and parallel efficiency with respect to the number of GPUs, indicating that both DG(P1) and rDG(P1P2) schemes could obtain high parallel efficiency even with the additional massage transfer required for implementing the periodic boundary conditions. However, the parallel efficiency is indeed lower than the previous 2D lid driven cavity problem, which also uses hexahedral elements but without the periodic boundary conditions. Nevertheless, the results obtained by GPU is still promising showing the capability and potential of the presented method for real-world problem.
Figure 7.23: Evolution of the dimensionless kinetic energy as a function of the dimensionless time using RK4.

Table 7.17: Timing measurements of using RK4+rDG(P_1P_2) methods for Taylor-Green vortex problem

<table>
<thead>
<tr>
<th>Nelem</th>
<th>T_{unit} (ms) by RK4 + rDG(P_1P_2)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64,000</td>
<td></td>
<td>4.45</td>
<td>48.58</td>
<td>10.9</td>
</tr>
<tr>
<td>531,441</td>
<td></td>
<td>3.01</td>
<td>55.91</td>
<td>18.6</td>
</tr>
</tbody>
</table>

Table 7.18: Timing measurements of weak scaling obtained on a cluster of Nvidia Tesla C2050 GPU cards for a Taylor-Green vortex problem.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Ngpu</th>
<th>T_{unit} (ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DG(P_1)</td>
<td>rDG(P_1P_2)</td>
</tr>
<tr>
<td>474,552</td>
<td>1</td>
<td>2.53</td>
<td>3.65</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2</td>
<td>2.64</td>
<td>3.97</td>
</tr>
<tr>
<td>2,000,376</td>
<td>4</td>
<td>2.84</td>
<td>4.31</td>
</tr>
<tr>
<td>3,944,312</td>
<td>8</td>
<td>2.86</td>
<td>4.36</td>
</tr>
</tbody>
</table>
Figure 7.24: Evolution of the dimensionless kinetic energy dissipation rate as a function of the dimensionless time using RK4.

Figure 7.25: Evolution of the dimensionless enstrophy rate as a function of the dimensionless time using RK4.
Figure 7.26: Plots of timing measurements for a 3D Taylor-Green Vortex with a fixed problem size (approximately 500,000 elements) per GPU card (Nvidia Tesla C2050): (a) Unit running time versus the number of GPU cards. (b) Parallel efficiency versus the number of GPU cards.
7.2 Model Equations

In this section, a number of numerical experiments for the model equations computed by the newly developed hyperbolic rDG methods are presented. Several steady model problems in a unit square are considered in this section. Several 2D cases would be computed on three types of grids. The first and second types are regular and irregular triangular grids with $9 \times 9$, $17 \times 17$, $33 \times 33$, and $65 \times 65$ nodes. The irregular grid is generated from the regular grid by random diagonal swapping and nodal perturbation. The third type is a set of heterogeneous grids with $12 \times 11$, $23 \times 21$, $45 \times 41$, and $99 \times 81$ nodes. The second grid of every type is shown in Figure 7.27.

In our work, the quality of these 3 types of grids is investigated as follows. For any triangle, there exists a ratio,

$$ r = \frac{R_{\text{out}}}{R_{\text{in}}}, $$

(7.12)

where, $R_{\text{out}}$ denotes the radius of the circumscribed circle, and $R_{\text{in}}$ refers to the radius of the inscribed circle. The ratio for the equilateral triangle, the best choice for numerical computation, is equal to 2.0. Therefore, in this work, the quality of meshes would be measured through

$$ R = \frac{r - 2}{2}. $$

(7.13)

![Figure 7.27](image)

Figure 7.27: The second mesh of every type, that is, $17 \times 17$ regular grid, $17 \times 17$ irregular grid, and $23 \times 21$ heterogeneous grid.

Clearly, larger $R$ indicates that the triangle meshes deviates greatly from the equilateral triangle. While the first type of mesh, i.e., the regular mesh, would remain same $R$ for the
entire computation domain, others may encounter large $R$. The contours of $R$ for the coarsest mesh from the irregular and heterogeneous mesh are shown in Figure 7.28. It could be observed that, for the first heterogeneous mesh, maximum of reached 80, which are almost 2 orders of magnitude higher than the one of the irregular mesh. This demonstrates the bad quality of the heterogeneous mesh, and also will demonstrate the robustness and accuracy of developed methods.

![Figure 7.28: Distribution of $R$ in the coarsest irregular mesh (left) and heterogeneous mesh (right).](image)

All the steady cases are computed with the initial solution set as 1.0 everywhere and the steady state is considered to be reached when the residual in $\varphi$ drops below $10^{-12}$ in the $L_2$ norm.

### 7.2.1 Steady Scalar Diffusion Case without Source Term

To begin with, a steady solution of the linear diffusion problem in a unit square is considered, i.e.,

$$\frac{\partial \varphi}{\partial t} = \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2}$$

(7.14)

with the exact solution given by

$$\varphi(x, y) = \frac{\sinh(\pi x) \sin(\pi y) \sinh(\pi y) \sin(\pi x)}{\sinh(\pi)}.$$  

(7.15)

The grid refinement study is carried out for regular, irregular and heterogeneous mesh to access the accuracy and the robustness of the presented hyperbolic rDG methods using BDF1.
$L_2$ norm of the difference between the numerical results and the exact ones is used as the error measurement. All norms are computed by Gaussian quadrature. The grid refinement study results for regular, irregular and heterogeneous meshes are shown in Fig. 7.29 to Fig. 7.31.

In all cases, the same order of accuracy, at least, has been achieved as expected. Note that the same order of accuracy is expected even when a higher-order polynomial is used in the primary variable; this is consistent with results for other hyperbolic schemes [114,119]. Exceptions are $\text{DG}(P_0P_1)+\text{DG}(P_0)$, $\text{DG}(P_0P_2)+\text{DG}(P_1)$, and $\text{DG}(P_0P_3)+r\text{DG}(P_1P_2)$, which achieve one-order higher accuracy in the primary variable. Such results have never been observed in the hyperbolic method before. Note, however, that $\text{DG}(P_0P_3)+r\text{DG}(P_1P_2)$ is not stable on the heterogeneous mesh. Therefore, WENO reconstruction has to be used to guarantee the stability of such scheme with the cost of achieving only 3rd order of accuracy for variable $\varphi$. Note that though upwind flux is implemented for all hyperbolic DGMs here, we use Rusanov flux scheme with $\text{DG}(P_0P_3)+r\text{DG}(P_1P_2)$ in irregular grids to compare. Clearly, Rusanov flux scheme would lead to less accurate results than upwind method due to too much dissipation, especially in $\varphi$.

Next, the comparison between hyperbolic DGMs and BR2 is carried out on regular, irregular and heterogeneous meshes using explicit TVDRK3, and the order of accuracy results are shown in Fig. 7.32 through Fig. 7.34. It is obvious that, $\text{DG}(P_1)$-BR2 could achieve 2nd order of accuracy for variable $\varphi$, and 1st order of accuracy for gradient of variable. Likewise, $\text{DG}(P_2)$-BR2 could achieve 3rd order of accuracy for variable $\varphi$, and 2nd order of accuracy for gradients. Note that the gradients are obtained always with one-order-lower accuracy than the primary variable, which is typical in conventional DGMs.

Several hyperbolic DGMs are presented to compare with BR2 scheme. Note that for heterogeneous grids, $\text{DG}(P_0P_3)+r\text{DG}(P_1P_2)$ is carried out with WENO reconstruction to remain stability. Results show that, our method is comparable to the BR2 method in terms of computational cost and accuracy. Note in particular that the same order of accuracy has been achieved by $\text{DG}(P_1)$-BR2 and $\text{DG}(P_0P_1)+\text{DG}(P_0)$, both of which involve exactly three discrete equations, and by $\text{DG}(P_2)$-BR2 and $\text{DG}(P_0P_2)+\text{DG}(P_1)$, which involve six and seven discrete equations, respectively.

Moreover, hyperbolic DG would also outperform BR2 in terms of the time steps and CPU time required to reach convergence. The simulation is carried out by hyperbolic DG method and BR2 method using TVDRK3 on the second mesh of each type, with the results shown in Fig. 7.35 to Fig. 7.34. Clearly, hyperbolic DGMs would be way less time consuming than standard BR2 scheme. As has been demonstrated for diffusion problems with other hyperbolic discretization methods [109,114,119], the hyperbolic method is known to achieve iterative con-
vergence acceleration by the elimination of numerical stiffness due to second-derivative diffusion operators.

To better illustrate the high efficiency of the developed hyperbolic rDG methods, Figure 7.38 shows the required total time steps and CPU time for hyperbolic rDG methods and traditional BR2 methods to reach the convergence on all each regular mesh while keeping the CFL number when refining the mesh. As expected, $O(h^2)$ time steps would be required for traditional diffusion scheme while the total iterations increase only linearly with the all the hyperbolic rDG methods. In other words, the presented methods lead to $O(1/h)$ acceleration in the steady convergence over traditional methods.

Figure 7.29: Grid refinement study of the hyperbolic rDG methods for the steady scalar diffusion case without source term on regular grids using implicit BDF1
Figure 7.30: Grid refinement study of the hyperbolic rDG methods for the steady scalar diffusion case without source term on regular grids using implicit BDF1.

Figure 7.31: Grid refinement study of the hyperbolic rDG methods for the steady scalar diffusion case without source term on heterogeneous grids using implicit BDF1.
Figure 7.32: Comparison between hyperbolic rDG methods and BR2 scheme in terms of order of accuracy for the steady scalar diffusion case without source term on regular grids using explicit TVDRK3.

Figure 7.33: Comparison between hyperbolic rDG methods and BR2 scheme in terms of order of accuracy for the steady scalar diffusion case without source term on irregular grids using explicit TVDRK3.
Figure 7.34: Comparison between hyperbolic rDG methods and BR2 scheme in terms of order of accuracy for the steady scalar diffusion case without source term on heterogeneous grids using explicit TVDRK3.

Figure 7.35: Comparison between hyperbolic rDG methods and BR2 scheme in terms of CPU time for the steady scalar diffusion case without source term on $17 \times 17$ regular grids using explicit TVDRK3.
Figure 7.36: Comparison between hyperbolic rDG methods and BR2 scheme in terms of CPU time for the steady scalar diffusion case without source term on 17 × 17 irregular grids using explicit TVDRK3.

Figure 7.37: Comparison between hyperbolic rDG methods and BR2 scheme in terms of CPU time for the steady scalar diffusion case without source term on 23 × 21 heterogeneous grids using explicit TVDRK3.
Figure 7.38: Time steps and CPU time required to reach the steady state for regular grids using explicit TVDRK3.
7.2.2 Steady Scalar Diffusion Case with Source Term

Another steady solution to a linear diffusion problem is considered in this case. The equation is given as

$$\frac{\partial \varphi}{\partial t} = \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} + f(x, y), \quad (7.16)$$

with the exact solution given by

$$\varphi(x, y) = 2 \cos(\pi x) \sin(2\pi y) + 2, \quad (7.17)$$

and

$$f(x, y) = 10\pi^2 \cos(\pi x) \sin(2\pi y). \quad (7.18)$$

Similarly, the simulations are performed in those three types of grids using hyperbolic DGMs with implicit BDF1. The numerical results are shown in Fig. 7.39 to Fig. 7.41.

The property of one-order higher accuracy in the primary variable for DG(P₀P₁)+DG(P₀) and DG(P₀P₂)+DG(P₁) is lost; but DG(P₀P₃)+rDG(P₁P₂) still gives fourth-order accuracy in $\varphi$ and third-order accuracy in the derivatives. Otherwise, the same order of accuracy is successfully achieved for all variables as expected.

Figure 7.39: Grid refinement study of the steady scalar diffusion case with source term on regular grids using implicit BDF1.
Figure 7.40: Grid refinement study of the steady scalar diffusion case with source term on irregular grids using implicit BDF1.

Figure 7.41: Grid refinement study of the steady scalar diffusion case with source term on heterogeneous grids using implicit BDF1.
7.2.3 Steady Tensor Diffusion Case with Source Term

To test the developed methods further, the following model diffusion problem is tested, i.e.,

\[
\frac{\partial \varphi}{\partial t} = \nabla \cdot (D \nabla \varphi) + f(x, y),
\]  

(7.19)

with the exact solution given by

\[
\varphi(x, y) = 1 - \tanh \left( \frac{(x - 0.5)^2 + (y - 0.5)^2}{0.01} \right),
\]  

(7.20)

and the diffusion tensor \( D \) is given by

\[
D = \begin{bmatrix}
(x + 1)^2 + y^2 & -xy \\
-x y & (y + 1)^2
\end{bmatrix}.
\]  

(7.21)

The source term can be computed as

\[
f = -\nabla \cdot (D \nabla \varphi) = 100(1 - A^2)(200AB + C),
\]  

(7.22)

where

\[
A = \tanh \left( \frac{(x - 0.5)^2 + (y - 0.5)^2}{0.01} \right),
\]  

(7.23)

\[
B = -4x^3(x + 1) + (4y^2 - 4y + 3)x^2 + 2(y + 1)x - 4y^3(y + 1) + 2(y^2 + y - 1),
\]  

(7.24)

and

\[
C = x(4x + 7) + y(6y + 7).
\]  

(7.25)

Again, grid refinement study is carried out for all three types of meshes using implicit BDF1, with the results shown in Fig. 7.42 to Fig. 7.44. Similar to the second case, \( \text{DG}(P_0P_1) + \text{DG}(P_0) \) and \( \text{DG}(P_0P_2) + \text{DG}(P_1) \) would not have super-convergence in \( \varphi \) while all other hyperbolic DGMs delivering designed order of accuracy for both variables and their derivatives, indicating the presented scheme is robust and can provide an attractive alternatives for using DGMs for diffusion problems.
Figure 7.42: Grid refinement study of the steady tensor diffusion case with source term on regular grids using implicit BDF1.

Figure 7.43: Grid refinement study of the steady tensor diffusion case with source term on irregular grids using implicit BDF1.
Figure 7.44: Grid refinement study of the steady tensor diffusion case with source term on heterogeneous grids using implicit BDF1.
7.2.4 Steady Nonlinear Diffusion Case

A steady model nonlinear diffusion problem in a unit square is considered in this section, i.e.,

\[
\frac{\partial \varphi}{\partial t} = \frac{\partial}{\partial x} \left( \nu(\varphi) \frac{\partial \varphi}{\partial x} \right) + \frac{\partial}{\partial y} \left( \nu(\varphi) \frac{\partial \varphi}{\partial y} \right) + f(x, y),
\]

with the exact solution, the diffusion coefficient and the source term given by

\[
\varphi(x, y) = \sin(\pi x) \sin(\pi y),
\]

\[
\nu = \nu(\varphi) = \varphi^2 + 1,
\]

and

\[
f(x, y) = 2\pi^2 \varphi \left( 3 \cos^2(\pi x) \cos^2(\pi y) - 2 \cos^2(\pi x) - 2 \cos^2(\pi y) + 2 \right).
\]

The grid refinement study has been carried out using the developed hyperbolic rDG methods on three sets of the meshes, namely regular, irregular and heterogeneous grids. The results are shown for each type of mesh in Table 7.19 and Fig. 7.45 to Fig. 7.47. Note that from this case, the standard basis matrix is adopted so that the degrees of freedom is same as the conventional DG methods. Besides the least-squares method, the variational reconstruction has been adopted starting this case as well.

![Grid refinement study](image)

Figure 7.45: Grid refinement study of the steady nonlinear diffusion case on regular grids using implicit BDF1.
Overall, the hyperbolic rDG methods delivered the designed order of accuracy for the case shown above. Note that several presented schemes like $\text{DG}(P_0P_3)+\text{rDG}_{-\text{LS}}(P_1P_2)$, $\text{DG}(P_0P_3)+\text{rDG}_{-\text{VR}}(P_0P_2)$, and $\text{DG}(P_0P_3)+\text{rDG}_{\text{VR}}(P_1P_2)$ are able to deliver 4th order in $\varphi$ and 3rd order in gradients in all the grids very effectively. Note that these results are better than expected since the hyperbolic scheme typically achieves the same order of accuracy (based on the lowest order of polynomials) for all variables, e.g., $\text{DG}(P_0P_{m+1})+\text{rDG}(P_0P_m)$ is expected to yield $(m+1)$-th order of accuracy for all variables. As for $\text{DG}(P_0P_4)+\text{rDG}_{\text{VR}}(P_1P_3)$ can obtain fourth order of accurate gradients. Additionally, one can find that variational reconstruction based rDG schemes are more stable than the least-squares rDG counterpart. Both $\text{DG}(P_0P_2)+\text{rDG}_{-\text{LS}}(P_0P_1)$ and $\text{DG}(P_0P_3)+\text{rDG}_{\text{LS}}(P_1P_2)$ are unable to deliver stable results for the finest heterogeneous grids without any limiter. On the contrary, the counterparts with variational reconstruction are stable and can deliver the desired order of accuracy for all the grids. Meanwhile, for variational reconstruction, one can have global stencil with compact data structure, thus to resolve the stability issue and make the extension to higher order reconstruction more straightforward. Also, boundary condition can be ignored for using variational reconstruction. The numerical results indicate that the presented hyperbolic rDG schemes are attractive and worth further investigation.

Figure 7.46: Grid refinement study of the steady nonlinear diffusion case on irregular grids using implicit BDF1.
Figure 7.47: Grid refinement study of the steady nonlinear diffusion case on heterogeneous grids using implicit BDF1.

Table 7.19: Order of accuracy on different type of grids for the steady nonlinear diffusion case.

<table>
<thead>
<tr>
<th></th>
<th>Regular grids</th>
<th>Irregular grids</th>
<th>Heterogeneous grids</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \varphi )</td>
<td>( v_x )</td>
<td>( \varphi )</td>
</tr>
<tr>
<td>DG(P0P1) + DG(P0)</td>
<td>0.93</td>
<td>1.00</td>
<td>0.93</td>
</tr>
<tr>
<td>DG(P0P2) + DG(P1)</td>
<td>1.93</td>
<td>1.88</td>
<td>1.80</td>
</tr>
<tr>
<td>DG(P0P3) + DG(P2)</td>
<td>3.76</td>
<td>2.73</td>
<td>3.84</td>
</tr>
<tr>
<td>DG(P0P2) + rDG_LS(P0P1)</td>
<td>2.28</td>
<td>2.07</td>
<td>2.14</td>
</tr>
<tr>
<td>DG(P0P3) + rDG_LS(P1P2)</td>
<td>3.83</td>
<td>3.02</td>
<td>3.84</td>
</tr>
<tr>
<td>DG(P0P2) + rDG_VR(P0P1)</td>
<td>1.97</td>
<td>1.99</td>
<td>1.96</td>
</tr>
<tr>
<td>DG(P0P3) + rDG_VR(P0P2)</td>
<td>3.94</td>
<td>3.35</td>
<td>4.02</td>
</tr>
<tr>
<td>DG(P0P3) + rDG_VR(P1P2)</td>
<td>3.83</td>
<td>3.02</td>
<td>3.76</td>
</tr>
<tr>
<td>DG(P0P4) + rDG_VR(P1P3)</td>
<td>4.02</td>
<td>3.85</td>
<td>4.01</td>
</tr>
</tbody>
</table>
7.2.5 1D Boundary Layer Case

In the first advection-diffusion test case, we consider the following 1D problem

\[ \frac{\partial \varphi}{\partial t} + a \frac{\partial \varphi}{\partial x} = \nu \frac{\partial^2 \varphi}{\partial x^2} + f(x), \quad 0 \leq x \leq 1, \]  
\[ \text{(7.30)} \]

with

\[ \varphi(0) = \varphi(1) = 1, \]  
\[ \text{(7.31)} \]

and the source term \( f(x) \) is given as

\[ f(x) = \frac{\pi}{\text{Re}} (a \cos(\pi x) + \pi \nu \sin(\pi x)), \quad \text{Re} = \frac{a}{\nu}. \]  
\[ \text{(7.32)} \]

The exact steady solution to the problem is

\[ \varphi(x) = \frac{e^\left(-\text{Re} - e^{(x\text{Re} - \text{Re})}\right)}{e^{(x\text{Re} - \text{Re})} - 1} + \frac{1}{\text{Re}} \sin(\pi x). \]  
\[ \text{(7.33)} \]

The exact solution can be regard as a function of Reynolds number. In the diffusion limit, it would be a smooth sine curve, while developing a very narrow boundary layer near \( x = 1 \) if advection limit is approached. Based on the FOHS formulation, we apply the developed hyperbolic rDG methods to solve the following equivalent system.

\[ \frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} = S, \]  
\[ \text{(7.34)} \]

where

\[ U = \left( \begin{array}{c} \varphi \\ v_x \end{array} \right), \quad F = \left( \begin{array}{c} a \varphi - \nu v_x \\ -\varphi/T_r \end{array} \right), \quad S = \left( \begin{array}{c} f(x) \\ -v_x/T_r \end{array} \right). \]  
\[ \text{(7.35)} \]

In order to capture the boundary layer, one would need enough resolution in the layer. Thus, numerical experiments are carried out with non-uniform grids generated from a uniform grid by the following mapping

\[ x_i = \frac{1 - e^{(i - \alpha \xi_i)}}{1 - \alpha}, \quad \xi_i = \frac{i - 1}{N_{\text{elem}}}. \]  
\[ \text{(7.36)} \]

For high Reynolds number case, or in other words, in the advection limit, one would need to increase \( \alpha \) to ensure the convergence. In this paper, we set \( a = 1 \), and all numerical results were obtained for a wide range of the Reynolds numbers, \( \text{Re} = 10^k \), where \( k = -8, 0, 8 \) with varying \( \nu \). And the corresponding \( \alpha \) is set to be 4.5, 4.5, 22.5. And the number of the elements
is set to be 32, 64, 128, and 256 for all Reynolds numbers.

Several hyperbolic rDG methods are applied here. For cases with smaller Reynolds numbers (Re = 1 and Re = 10\(^{-8}\)), all presented method achieve designed or even higher order of accuracy. See Table 7.20. However, for high Reynolds number case, a very strong boundary layer arises near \(x = 1\), and some hyperbolic methods, including those in Ref. [104], become unstable. These results are somewhat consistent with those in Ref. [118]. The presented methods are in the same family of the Scheme II in Ref. [118], which directly uses the gradient variables to construct higher order polynomials for the primary solution variable. Ref. [118] shows that this efficient construction introduces an artificial negative diffusion coefficient for a high-Reynolds-number boundary-layer-type problem, resulting accuracy and convergence problems [118]. However, the first-order scheme, and our target schemes based on \(\text{rDG}(P_0P_k)\) with \(k = 1, 2\) do not suffer such a problem and are found to be stable, delivering their design order of accuracy. Unstable schemes may be remedied by extending techniques suggested in Ref. [118], e.g., \(\text{DG}(P_n)+\text{DG}(P_n)\) or \(\text{rDG}(P_nP_m)+\text{rDG}(P_nP_m)\) [81], the upwind flux based on the unified eigen-structure, or indirect gradient reconstructions called Scheme IQ in the Ref. [118]. These approaches should be explored in future work.

Table 7.20: Order of accuracy for the 1D boundary layer case with different Re.

<table>
<thead>
<tr>
<th>Scheme [DoFs]</th>
<th>Advection (\text{Re} = 10^8)</th>
<th>Advection-Diffusion (\text{Re} = 1)</th>
<th>Diffusion (\text{Re} = 10^{-8})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{DG}(P_0P_1)+\text{DG}(P_0)) [2]</td>
<td>0.91 0.98</td>
<td>1.00 1.00</td>
<td>1.01 1.00</td>
</tr>
<tr>
<td>(\text{DG}(P_0P_2)+\text{DG}(P_1)) [3]</td>
<td>- -</td>
<td>2.00 2.00</td>
<td>2.00 2.00</td>
</tr>
<tr>
<td>(\text{DG}(P_0P_3)+\text{DG}(P_2)) [4]</td>
<td>- -</td>
<td>4.01 3.00</td>
<td>3.96 3.00</td>
</tr>
<tr>
<td>(\text{DG}(P_0P_2)+\text{rDG}_{LS}(P_0P_1)) [2]</td>
<td>2.00 2.14</td>
<td>1.97 2.05</td>
<td>1.97 2.04</td>
</tr>
<tr>
<td>(\text{DG}(P_0P_3)+\text{rDG}_{LS}(P_1P_2)) [3]</td>
<td>- -</td>
<td>3.12 2.68</td>
<td>3.11 2.69</td>
</tr>
<tr>
<td>(\text{DG}(P_0P_2)+\text{rDG}_{VR}(P_0P_1)) [2]</td>
<td>2.00 1.90</td>
<td>2.00 2.05</td>
<td>2.00 2.05</td>
</tr>
<tr>
<td>(\text{DG}(P_0P_3)+\text{rDG}_{VR}(P_0P_2)) [2]</td>
<td>3.96 3.02</td>
<td>3.99 3.69</td>
<td>4.06 3.66</td>
</tr>
<tr>
<td>(\text{DG}(P_0P_3)+\text{rDG}_{VR}(P_1P_2)) [3]</td>
<td>- -</td>
<td>3.89 3.06</td>
<td>3.80 3.04</td>
</tr>
</tbody>
</table>
Figure 7.48: Grid refinement study for the 1D boundary layer case, $Re = 10^{-8}$.

Figure 7.49: Grid refinement study for the 1D boundary layer case, $Re = 1$. 

162
Figure 7.50: Grid refinement study for the 1D boundary layer case, $Re=10^8$. 
7.2.6 2D Steady Advection-Diffusion Case

A steady model advection diffusion problem in a unit square is considered in this section, i.e.,

\[
\frac{\partial \varphi}{\partial t} + a \frac{\partial \varphi}{\partial x} + b \frac{\partial \varphi}{\partial y} = \nu \left( \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} \right),
\]

(7.37)

with the exact solution given by

\[
\varphi(x, y) = C \cos(A \pi \eta) \exp \left( \frac{1 - \sqrt{1 + 4A^2 \pi^2 \nu^2}}{2\nu} \xi \right),
\]

(7.38)

and

\[
\xi = ax + by, \quad \eta = bx - ay.
\]

(7.39)

In this case, we take \((a, b) = (2, 1), A = 2, C = -0.009\) with \(\nu = 10^{-8}, 10^0, 10^8\) to test the convergence rate for different hyperbolic rDG methods. Three sets of meshes are used in the test, namely regular, irregular and heterogeneous grids. The sample of each type of grids are shown in Figure 7.27.

The grid refinement study has been carried out for the hyperbolic rDG methods. In each type of mesh, the advection limit case \((\nu = 10^{-8})\), advection-diffusion case \((\nu = 1)\), and the diffusion limit case \((\nu = 10^8)\) are studied, with the results shown in Tables 7.21 to 7.23 and Figures 7.51 to 7.59.

Overall, the hyperbolic rDG methods were able to deliver the designed or higher order of accuracy for most of the cases. Note that there exists no boundary layer in this problem, and thus all schemes converged without any problem in the advection limit. However, we do observe that \(\text{DG}(P_0P_2) + \text{rDG}_\text{LS}(P_0P_1)\) being unstable for non-advection limit case. It appears that this issue would be fixed by either adding more cells in the LS stencil or applying limiters. On the other hand, \(\text{DG}(P_0P_3) + \text{rDG}_\text{LS}(P_1P_2), \text{DG}(P_0P_3) + \text{rDG}_\text{VR}(P_0P_2), \) and \(\text{DG}(P_0P_3) + \text{rDG}_\text{-VR}(P_1P_2)\) are able to deliver fourth-order accuracy in \(\varphi\) and third-order accuracy in gradients in all the cases very effectively. Variational reconstruction is based on a global stencil with compact data structure, resolving the stability issue of the LS reconstruction and thus making the extension to higher order reconstruction more straightforward. Also, the variational reconstruction can be performed without taking into account boundary conditions. The numerical results indicate that the presented hyperbolic rDG schemes with the variational reconstruction are attractive and worth further investigation.

Further more, we ported the code onto GPU platform based on OpenACC, and a strong
Table 7.21: Order of accuracy for the 2D steady advection-diffusion case on regular grids with different $\nu$.

<table>
<thead>
<tr>
<th>Scheme [DoFs]</th>
<th>Advection $\nu = 10^{-8}$</th>
<th>Advection-Diffusion $\nu = 1$</th>
<th>Diffusion $\nu = 10^{8}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\varphi$</td>
<td>$v_x$</td>
<td>$\varphi$</td>
</tr>
<tr>
<td>DG(P_0P_1)+DG(P_0) [3]</td>
<td>2.11</td>
<td>0.99</td>
<td>1.26</td>
</tr>
<tr>
<td>DG(P_0P_2)+DG(P_1) [6]</td>
<td>3.02</td>
<td>2.01</td>
<td>2.03</td>
</tr>
<tr>
<td>DG(P_0P_3)+DG(P_2) [10]</td>
<td>3.97</td>
<td>2.97</td>
<td>3.65</td>
</tr>
<tr>
<td>DG(P_0P_2)+rDG_LS(P_0P_1) [3]</td>
<td>3.18</td>
<td>2.01</td>
<td>-</td>
</tr>
<tr>
<td>DG(P_0P_3)+rDG_LS(P_1P_2) [6]</td>
<td>4.14</td>
<td>3.22</td>
<td>3.69</td>
</tr>
<tr>
<td>DG(P_0P_2)+rDG_VR(P_0P_1) [3]</td>
<td>3.11</td>
<td>2.00</td>
<td>2.88</td>
</tr>
<tr>
<td>DG(P_0P_3)+rDG_VR(P_0P_2) [3]</td>
<td>4.49</td>
<td>3.28</td>
<td>3.04</td>
</tr>
<tr>
<td>DG(P_0P_3)+rDG_VR(P_1P_2) [6]</td>
<td>4.30</td>
<td>3.06</td>
<td>3.74</td>
</tr>
</tbody>
</table>

scaling test is deduced. Note that the triangular mesh is then extended to a prismatic mesh with only 1 cell in $z$ direction to perform the quasi-2D test case in a general 3D code framework. The detailed timing measurements are presented in Table 7.24, Table 7.24, and Table 7.26 showing the statistics of unit running time. Clearly, the GPU accelerated hyperbolic rDG methods has shown similar and even better pattern compared with those flow simulations on GPU, indicating the great potential for extending the hyperbolic rDG method to solve the full Navier-Stokes equations on the GPU platforms.
Table 7.22: Order of accuracy for the 2D steady advection-diffusion case on irregular grids with different $\nu$.

<table>
<thead>
<tr>
<th>Scheme [DoFs]</th>
<th>Advection $\nu = 10^{-8}$</th>
<th>Advection-Diffusion $\nu = 1$</th>
<th>Diffusion $\nu = 10^{8}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\varphi$</td>
<td>$v_x$</td>
<td>$\varphi$</td>
</tr>
<tr>
<td>DG($P_0P_1$)+DG($P_0$) [3]</td>
<td>1.93</td>
<td>0.99</td>
<td>1.26</td>
</tr>
<tr>
<td>DG($P_0P_2$)+DG($P_1$) [6]</td>
<td>2.74</td>
<td>1.91</td>
<td>2.38</td>
</tr>
<tr>
<td>DG($P_0P_3$)+DG($P_2$) [10]</td>
<td>3.97</td>
<td>2.97</td>
<td>2.99</td>
</tr>
<tr>
<td>DG($P_0P_2$)+rDG_LS($P_0P_1$) [3]</td>
<td>2.80</td>
<td>1.92</td>
<td>-</td>
</tr>
<tr>
<td>DG($P_0P_3$)+rDG_LS($P_1P_2$) [6]</td>
<td>4.18</td>
<td>3.19</td>
<td>3.61</td>
</tr>
<tr>
<td>DG($P_0P_2$)+rDG_VR($P_0P_1$) [3]</td>
<td>2.74</td>
<td>1.93</td>
<td>2.78</td>
</tr>
<tr>
<td>DG($P_0P_3$)+rDG_VR($P_0P_2$) [3]</td>
<td>3.77</td>
<td>2.97</td>
<td>3.02</td>
</tr>
<tr>
<td>DG($P_0P_3$)+rDG_VR($P_1P_2$) [6]</td>
<td>3.87</td>
<td>3.01</td>
<td>3.97</td>
</tr>
</tbody>
</table>

Table 7.23: Order of accuracy for 2D steady advection-diffusion case on heterogeneous grids with different $\nu$.

<table>
<thead>
<tr>
<th>Scheme [DoFs]</th>
<th>Advection $\nu = 10^{-8}$</th>
<th>Advection-Diffusion $\nu = 1$</th>
<th>Diffusion $\nu = 10^{8}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\varphi$</td>
<td>$v_x$</td>
<td>$\varphi$</td>
</tr>
<tr>
<td>DG($P_0P_1$)+DG($P_0$) [3]</td>
<td>2.11</td>
<td>0.95</td>
<td>1.17</td>
</tr>
<tr>
<td>DG($P_0P_2$)+DG($P_1$) [6]</td>
<td>3.15</td>
<td>2.07</td>
<td>2.60</td>
</tr>
<tr>
<td>DG($P_0P_3$)+DG($P_2$) [10]</td>
<td>4.08</td>
<td>3.04</td>
<td>3.12</td>
</tr>
<tr>
<td>DG($P_0P_2$)+rDG_LS($P_0P_1$) [3]</td>
<td>3.05</td>
<td>2.06</td>
<td>-</td>
</tr>
<tr>
<td>DG($P_0P_3$)+rDG_LS($P_1P_2$) [6]</td>
<td>4.17</td>
<td>3.11</td>
<td>3.87</td>
</tr>
<tr>
<td>DG($P_0P_2$)+rDG_VR($P_0P_1$) [3]</td>
<td>3.05</td>
<td>2.06</td>
<td>2.38</td>
</tr>
<tr>
<td>DG($P_0P_3$)+rDG_VR($P_0P_2$) [3]</td>
<td>4.51</td>
<td>3.27</td>
<td>3.57</td>
</tr>
<tr>
<td>DG($P_0P_3$)+rDG_VR($P_1P_2$) [6]</td>
<td>4.23</td>
<td>3.08</td>
<td>3.89</td>
</tr>
</tbody>
</table>
Figure 7.51: Grid refinement study for the 2D steady advection-diffusion case on regular grids with $\nu = 10^{-8}, \text{Re} = \sqrt{5} \times 10^8$.

Figure 7.52: Grid refinement study for the 2D steady advection-diffusion case on regular grids with $\nu = 1, \text{Re} = \sqrt{5}$. 
Figure 7.53: Grid refinement study for the 2D steady advection-diffusion case on regular grids with $\nu = 10^8$, $Re = \sqrt{5} \times 10^8$.

Figure 7.54: Grid refinement study for the 2D steady advection-diffusion case on irregular grids with $\nu = 10^{-8}$, $Re = \sqrt{5} \times 10^8$. 

168
Figure 7.55: Grid refinement study for the 2D steady advection-diffusion case on irregular grids with $\nu = 1, \text{Re} = \sqrt{5}$.

Figure 7.56: Grid refinement study for the 2D steady advection-diffusion case on irregular grids with $\nu = 10^8, \text{Re} = \sqrt{5} \times 10^{-8}$. 

169
Figure 7.57: Grid refinement study for the 2D steady advection-diffusion case on heterogeneous grids with $\nu = 10^{-8}, \text{Re} = \sqrt{5} \times 10^8$.

Figure 7.58: Grid refinement study for the 2D steady advection-diffusion case on heterogeneous grids with $\nu = 1, \text{Re} = \sqrt{5}$. 
Figure 7.59: Grid refinement study for the 2D steady advection-diffusion case on heterogeneous grids with $\nu = 10^8$, $Re = \sqrt{5} \times 10^{-8}$.

Table 7.24: Timing measurements of using hyperbolic DG/rDG methods for the quasi-2D steady advection-diffusion case on regular grids.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{unit}$ (ms) by DG($P_0P_1$)+DG($P_0$)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
<th>$T_{unit}$ (ms) by DG($P_0P_2$)+rDG$_{LS}$($P_0P_1$)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8.96</td>
<td>11.72</td>
<td>1.31</td>
<td></td>
<td>15.00</td>
<td>12.50</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>2.45</td>
<td>11.39</td>
<td>4.65</td>
<td></td>
<td>4.32</td>
<td>12.04</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>2,048</td>
<td>0.74</td>
<td>11.38</td>
<td>15.33</td>
<td></td>
<td>1.18</td>
<td>11.91</td>
<td>10.05</td>
<td></td>
</tr>
<tr>
<td>8,192</td>
<td>0.47</td>
<td>11.38</td>
<td>24.27</td>
<td></td>
<td>0.58</td>
<td>12.03</td>
<td>20.76</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.25: Timing measurements of using hyperbolic DG/rDG methods for the quasi-2D steady advection-diffusion case on irregular grids.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{unit}$ by DG($P_0P_1$)+DG($P_0$)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
<th>$T_{unit}$ by DG($P_0P_2$)+rDG$_{LS}$($P_0P_1$)</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>9.58</td>
<td>11.98</td>
<td>1.25</td>
<td></td>
<td>15.21</td>
<td>12.50</td>
<td>0.82</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>2.45</td>
<td>11.39</td>
<td>4.65</td>
<td></td>
<td>4.32</td>
<td>12.04</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>2,048</td>
<td>0.78</td>
<td>11.41</td>
<td>14.60</td>
<td></td>
<td>1.21</td>
<td>11.93</td>
<td>9.85</td>
<td></td>
</tr>
<tr>
<td>8,192</td>
<td>0.49</td>
<td>11.38</td>
<td>23.46</td>
<td></td>
<td>0.59</td>
<td>12.03</td>
<td>20.42</td>
<td></td>
</tr>
</tbody>
</table>
Table 7.26: Timing measurements of using hyperbolic DG/rDG methods for the quasi-2D steady advection-diffusion case on heterogeneous grids.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>$T_{\text{unit}}$ by DG($P_0P_1$)+DG($P_0$)</th>
<th>$T_{\text{unit}}$ by DG($P_0P_2$)+rDG,LS($P_0P_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td>440</td>
<td>2.79</td>
<td>11.44</td>
</tr>
<tr>
<td>1,760</td>
<td>0.83</td>
<td>11.42</td>
</tr>
<tr>
<td>7,040</td>
<td>0.47</td>
<td>11.35</td>
</tr>
<tr>
<td>28,160</td>
<td>0.39</td>
<td>11.44</td>
</tr>
</tbody>
</table>
7.2.7 1D Unsteady Advection-Diffusion Case

In this section, the 1D unsteady advection-diffusion problem is presented as

\[
\frac{\partial \phi}{\partial t} + a \frac{\partial \phi}{\partial x} = \nu \frac{\partial^2 \phi}{\partial x^2}, \quad 0 \leq x \leq 2,
\]

with the following exact solution,

\[
\phi(x,t) = \frac{1}{\sqrt{4t + 1}} \exp \left( -\frac{(x - at - x_0)^2}{\nu(4t + 1)} \right),
\]

where

\[
a = 10^4, \quad \nu = 0.01, \quad x_0 = 0.5.
\]

The parameters are chosen to yield an advection dominant problem. The initial Gaussian bump will travel with a constant velocity with a small diffusion effect. In this paper, a grid refinement test has been carried out to verify the spatial order of accuracy in this unsteady case. A small physical time step \( \Delta t = 10^{-9} \) has been set for all the grids with a fixed final time \( t_{\text{end}} = 10^{-4} \). The grids are uniform with \( nelem = 32, 64, 128 \) and 256. Dirichlet boundary conditions has been applied on both ends. All the presented methods use the same degrees of freedom, 2, which is equivalent to a conventional P_1 DG method. The numerical results are shown in Figure 7.60.

As we can see, all the presented hyperbolic rDG methods can deliver the design or higher order of accuracy for the unsteady problem.

Secondly, a comparison between the developed schemes and a conventional DG (Direct DG in this study), is shown in Figure 7.61. Periodic boundary conditions are enforced with \( \Delta t = 10^{-9}, \ t_{\text{end}} = 10^{-3} \) on a uniform mesh (\( nelem = 32 \)). Clearly, the presented hyperbolic rDG methods can outperform the conventional counterpart, better resolving the peak of the Gaussian profile.
Figure 7.60: Grid refinement study on regular grids for the 1D unsteady advection-diffusion case.

Figure 7.61: Comparison between the numerical solutions obtained by the hyperbolic rDG methods and DDG methods for the 1D unsteady advection-diffusion case, $t_{end} = 10^{-3}$ (5 periods).
7.2.8 2D Unsteady Advection-Diffusion Case

In this case, a 2D unsteady case are considered.

\[
\frac{\partial \varphi}{\partial t} + a \frac{\partial \varphi}{\partial x} + b \frac{\partial \varphi}{\partial y} = \nu \left( \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} \right), \quad (x, y) \in [0, 2] \times [0, 2],
\]

(7.43)

The analytical solution is given as

\[
\varphi(x, y, t) = \frac{1}{4t + 1} \exp \left( -\frac{(x - at - x_0)^2 + (y - bt - y_0)^2}{\nu(4t + 1)} \right),
\]

(7.44)

where

\[
x_0 = y_0 = 1.0, \quad a = b = 10^{-5}, \quad \nu = 0.01.
\]

(7.45)

The regular and irregular grids of the 2D steady case are applied here with a scaling factor of 2. The physical time step is set as \(\Delta t = 10^{-3}\) with \(t_{end} = 1\). Dirichlet boundary conditions are applied on all the boundary faces. Here, several hyperbolic rDG schemes are presented here, including \(\text{DG}(P_0P_1) + \text{DG}(P_0)\), \(\text{DG}(P_0P_2) + \text{DG}(P_1)\), \(\text{DG}(P_0P_2) + \text{rDG}_\text{LS}(P_0P_1)\), \(\text{DG}(P_0P_3) + \text{rDG}_\text{LS}(P_1P_2)\), \(\text{DG}(P_0P_2) + \text{rDG}_\text{VR}(P_0P_1)\), \(\text{DG}(P_0P_3) + \text{rDG}_\text{VR}(P_0P_2)\), \(\text{DG}(P_0P_3) + \text{rDG}_\text{VR}(P_1P_2)\), \(\text{DG}(P_0P_4) + \text{rDG}_\text{VR}(P_1P_3)\). The numerical results are shown in Table 7.27 and Figures 7.62 to 7.63. All the presented schemes are shown to provide the design or higher order of accuracy in this unsteady case.

Figure 7.62: Grid refinement study on regular grids for 2D unsteady advection-diffusion case.
Figure 7.63: Grid refinement study on irregular grids for 2D unsteady advection-diffusion case.

Table 7.27: Order of accuracy for the 2D unsteady advection-diffusion case.

<table>
<thead>
<tr>
<th>Scheme [DoFs]</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\varphi$</td>
<td>$v_x$</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_1) + \text{DG}(P_0)$</td>
<td>0.97</td>
<td>0.79</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_2) + \text{DG}(P_1)$</td>
<td>1.96</td>
<td>1.86</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_2) + r\text{DG}_{LS}(P_0P_1)$</td>
<td>2.34</td>
<td>2.24</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_3) + r\text{DG}_{LS}(P_1P_2)$</td>
<td>3.96</td>
<td>3.34</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_2) + r\text{DG}_{VR}(P_0P_1)$</td>
<td>2.89</td>
<td>2.30</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_3) + r\text{DG}_{VR}(P_0P_2)$</td>
<td>5.30</td>
<td>4.57</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_3) + r\text{DG}_{VR}(P_1P_2)$</td>
<td>3.76</td>
<td>3.66</td>
</tr>
<tr>
<td>$\text{DG}(P_0P_4) + r\text{DG}_{VR}(P_1P_3)$</td>
<td>5.14</td>
<td>5.06</td>
</tr>
</tbody>
</table>
Chapter 8

Conclusions

8.1 Summary of Completed Work

An OpenACC directive-based parallel scheme has been designed and developed for the GPU computing of RDGFLO, a 3D CFD Fortran code for compressible flows at all speeds on hybrids unstructured grids through a synergistic hardware/software co-design approach. The solver is implemented with rDG(P₁P₂), a third-order reconstructed discontinuous Galerkin method based on a Hierarchical WENO reconstruction, designed not only to enhance the accuracy of the underlying second-order DG(P₁) method but also to avoid non-physical oscillations in the vicinity of strong discontinuities, without significant increase in computing costs and the memory requirements. Indeed, compared with the more mature and dominating techniques like CUDA, the current OpenACC specification and compilers have not yet been well defined and optimized although active development and improvement are underway. Therefore, it is not surprising that a fine-tuned CUDA code could usually outperform the equivalent OpenACC code as of today. Nevertheless, as we have stressed, the biggest benefits of adopting OpenACC for our CFD solvers are still evident: it requires the minimum code intrusion and algorithm alteration to upgrade a legacy unstructured CFD solver with the GPU computing capability without much extra effort in programming, thus could save tremendous work hours in code development and maintenance.

Besides, a new family of hyperbolic reconstructed Discontinuous Galerkin methods has been developed based on the first order hyperbolic system (FOHS). Unlike the conventional DG methods for the diffusive flux, hyperbolic rDG methods generate a system of pseudo-time evolution equations for the solution and the derivatives in the partial differential equation level instead
of in the discretization level. This would allow a dramatic simplification in the discretization since the well-established methods for hyperbolic systems can be directly applied to the viscous terms. Other than the least squares reconstruction, a newly developed reconstruction scheme, variational reconstruction has also been implemented in the hyperbolic rDG methods, providing arbitrary higher-order numerical solutions robustly while remaining not only the total degrees of freedom relatively small but also the data structure compact. By combining the advantages of the FOHS formulations and the rDG schemes, an effort has been made to develop more reliable, accurate, efficient and robust methods to deal with the diffusive fluxes.

For the time advancement, both multi-stage explicit Runge-Kutta methods and the simple implicit backward Euler methods are implemented in the GPU accelerated RDGFLO solver. Additionally, for the unsteady case solved by the newly presented hyperbolic rDG methods, a dual time stepping technique has been applied, while the ESDIRK scheme is used for the physical temporal discretization, providing a robust high order implicit time scheme with varying time-steps. Moreover, $p$-multigrid technique is also adopted in the study on the GPU platform to further accelerate the convergence. The Jacobian matrix in the implicit methods is obtained through the analytical differentiation and automatic differentiation. The resulting linear system is solved using the general minimum residual (GMRES) algorithm with lower-upper symmetric Gauss-Seidel (LU-SGS) preconditioning on the CPU and symmetric Gauss-Seidel (SGS) methods on the GPU. Due to the fact that GPGPU is a share memory device, one would need to consider the memory contention issue for the implementation on it. A face coloring algorithm is employed to eliminate the “race condition” from the threading of the face integrals. A similar element reordering algorithm needs to be adopted to resolve the inherent data dependency for implementing LU-SGS or SGS preconditioner/solver. Also, a fine-grained style algorithm for Gauss-Jordan elimination is used for the efficient matrix inversion on GPU platform. For large-scale problems of practical importance, a message passing interface (MPI) programming paradigm is used to enable the multi-GPU computing ability, while METIS library is employed for the partitioning of a mesh into subdomain meshes of approximately the same size.

A series of inviscid and viscous flow problems, including the large eddy simulation type (“lid driven cavity”) and DNS type (“Taylor Green vortex”) are conducted to verify the implementation of the developed scheme on GPU platform. The numerical experiments form the verification tests, strong scaling tests and weak scaling tests indicate that this OpenACC based parallel scheme is able to significantly accelerate the solving for the equivalent legacy CPU code with high parallel efficiency. Meanwhile, the results for the model equations for the newly developed hyperbolic rDG methods demonstrate the presented schemes are able to achieve the designed optimal order of accuracy for both solutions and their derivatives on regular, irregular
and heterogeneous grids, and outperform the conventional diffusive DG methods like BR2, DDG, in terms of the magnitude of the error, the order of accuracy, the size of time steps, and the CPU times required to achieve pseudo steady state solutions, indicating that the developed hyperbolic rDG methods provide an attractive and probably an even superior alternative to deal with diffusive fluxes.

In summary, the developed GPU accelerated rDG method and the hyperbolic rDG schemes have been assessed and validated, showing great potential for further investigation. The work accomplished in this Ph.D. work only represents a first step towards the development of an accurate, efficient, and robust hyperbolic rDG code to simulate 3D aerodynamic flows on arbitrary grids on both CPU and GPU platforms.

8.2 Outlook of Future Work

The future development of this work towards RDGFLO code may be focused on two main directions: one direction is to tap the full potential of the GPU devices and thus to optimize the code for better performance on GPU using OpenACC directives; the other can be the further development of the hyperbolic rDG methods to full Navier-Stokes equations.

For the first part, the high order implicit time integration methods, like implicit Runge-Kutta methods (ESDIRK methods and Rosenbrock-Wanner methods (ROW) methods) are expected to be implemented using OpenACC as well to simulate the time-accurate flow problems implicitly on the GPUs. Currently, the GMRES linear solver has not been ported onto the GPU platform due to the limitation of the memory of GPU card, which would need to be taken care of in future. Meanwhile, the current bottleneck for the implicit method is that the computation of the Jacobian matrix on the GPU platform is highly memory bounded, leading to small speedup factors in the GPU devices. One would need to either design a fine-grained style algorithm for those subroutines on the GPU platforms or consider the Jacobian-free type implicit solver to tap the full performance. Also, one can add the turbulence model to the existing GPU accelerated to generate the RANS system for the 3D steady compressible turbulent problem. While the \( p \)-adaptivity is enabled on the GPU platform, the \( h \)-adaptivity could also be the further work of the development of the RDGFLO.

As for the second part, efforts are being taken to generate a hyperbolic Navier-Stokes equations system using first order hyperbolic system formulation to extend the current work of hyperbolic rDG methods to real computational fluid dynamics region. Afterwards, one may consider to port the hyperbolic rDG Navier-Stokes methods on the GPU platform to enable the cross-platform ability.
REFERENCES


186


APPENDICES
In this appendix, the equivalence of the variational reconstruction to the well-known compact finite difference scheme is demonstrated on a 1D uniform grid. Consider to perform variational reconstruction $P_0P_1$ on the uniform grid. The jump at interface $i + 1/2$ is defined as

$$I_{i+1/2} = \frac{1}{2} \left\{ \left[ w_0 (u_{i+1/2}^+ - u_{i+1/2}^-) \right]^2 + \left[ w_1 h \left( \frac{du}{dx}_{i+1/2}^+ - \frac{du}{dx}_{i+1/2}^- \right) \right]^2 \right\}$$  \hspace{1cm} (A.1)$$

where

$$u_{i+1/2}^+ = u_{i+1} - \frac{h}{2} s_{i+1}, \quad u_{i+1/2}^- = u_i + \frac{h}{2} s_i, \quad \frac{du}{dx}_{i+1/2}^+ = s_{i+1}, \quad \text{and} \quad \frac{du}{dx}_{i+1/2}^- = s_i. \hspace{1cm} (A.2)$$

The objective function is defined as the summation of the interface jumps over all the interfaces

$$I = \sum_{i=0}^{n} I_{i+1/2}. \hspace{1cm} (A.3)$$

The equations for the slopes $s$ can be derived by finding the minimizer of the objective

$$194$$
function. Taking the derivative $I$ with respect to $s_i$ and setting it to be zero lead to
\[
\frac{\partial I}{\partial s_i} = \sum_{j=0}^{n} \frac{\partial I_{j+1/2}}{\partial s_i} = \frac{\partial I_{i-1/2}}{\partial s_i} + \frac{\partial I_{i+1/2}}{\partial s_i} = 0 \tag{A.4}
\]

Note that only $I_{i-1/2}$ and $I_{i+1/2}$ contain the coefficient $s_i$ and
\[
\frac{\partial I_{i+1/2}}{\partial s_i} = w_0(u_{i+1/2}^+ - u_{i+1/2}^-) \left( -\frac{w_0 h}{2} \right) + w_1 h \left( \frac{du}{dx}_{i+1/2}^+ - \frac{du}{dx}_{i+1/2}^- \right) (-w_1 h) \\
= \left( \frac{w_0^2}{4} + w_1^2 \right) h^2 s_i + \left( \frac{w_0^2}{4} - w_1^2 \right) h^2 s_{i+1} - \frac{w_0^2 h}{2} (u_{i+1} - u_i) \tag{A.5}
\]
\[
\frac{\partial I_{i-1/2}}{\partial s_i} = w_0(u_{i-1/2}^+ - u_{i-1/2}^-) \left( -\frac{w_0 h}{2} \right) + w_1 h \left( \frac{du}{dx}_{i-1/2}^+ - \frac{du}{dx}_{i-1/2}^- \right) (w_1 h) \\
= \left( \frac{w_0^2}{4} + w_1^2 \right) h^2 s_{i-1} + \left( \frac{w_0^2}{4} - w_1^2 \right) h^2 s_i - \frac{w_0^2 h}{2} (u_i - u_{i-1}) \tag{A.6}
\]

The resulting linear equation for the slope $s$ becomes
\[
\left( \frac{w_0^2}{4} - w_1^2 \right) h^2 s_{i-1} + 2 \left( \frac{w_0^2}{4} + w_1^2 \right) h^2 s_i + \left( \frac{w_0^2}{4} - w_1^2 \right) h^2 s_{i+1} = \frac{w_0^2 h}{2} (u_{i+1} - u_{i-1}) \tag{A.7}
\]

Note that if $w_1 = 0.5w_0$, the slope is then given by the central differencing scheme, i.e.,
\[
s_i = \frac{u_{i+1} - u_{i-1}}{2h}. \tag{A.8}
\]

When $w_1 = 0$, the equation would reduce to
\[
\frac{s_{i-1}}{2} + s_i + \frac{s_{i+1}}{2} = \frac{u_{i+1} - u_{i-1}}{h}, \tag{A.9}
\]
which is nothing but the second-order compact finite difference scheme.

Consequently, the variational reconstruction scheme can be regarded as the extension of the compact finite difference scheme [70] on unstructured grids. The resultant linear system for the slopes is symmetric, compact, and diagonally dominant. When $w_1$ is not equally to zero, the matrix of the linear system is strictly diagonally dominant, and therefore nonsingular. Thus, the gradient solution always exists and is unique and easy to compute numerically.
Appendix B

Various Types of Elements

In this section, the shape functions of the various reference elements frequently used in practice are presented. The geometrical types of elements are listed in Table B.1. From the geometrical information, we can construct the shape functions $\phi(\xi, \eta, \zeta)$ and their derivatives with respect to $\xi$, $\eta$ and $\zeta$. A more comprehensive introduction to the various types of elements are given in §2 of the book *Finite Element Method* by Dhatt et al. [36].

B.1 Triangular Elements (two dimensions)

B.1.1 Systems of Coordinates

For all triangular elements, the following reference element shown in Fig. B.1 is used, where $\xi \leq 0$, $\eta \leq 0$ and $1 - \xi - \eta \leq 0$. The coordinates $(\xi, \eta)$ can be interpreted as curvilinear coordinates for the physical element. The barycentric coordinates $L_1$, $L_2$ and $L_3$ are often used to mark a point $0$ of a straight-edged triangle as shown in Fig. B.2.

\[
L_1 = \frac{A_1}{A}; \quad L_2 = \frac{A_2}{A}; \quad L_3 = \frac{A_3}{A},
\]

\[
A = A_1 + A_2 + A_3,
\]

\[
L_1 + L_2 + L_3 = 1,
\]

where $A_1$, $A_2$ and $A_3$ are the areas of the subtriangles 0-2-3, 0-3-1 and 0-1-2. $A$ is the area of the triangle 1-2-3. The coordinates $L_1$, $L_2$ and $L_3$ are linked to the coordinates $(\xi, \eta)$ by the following relation:

\[
L_1 \equiv 1 - \xi - \eta, \quad L_2 \equiv \xi, \quad L_3 \equiv \eta.
\]
Table B.1: Abbreviation and description for various types of elements.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIA3</td>
<td>3-node triangle.</td>
</tr>
<tr>
<td>TRIA6</td>
<td>6-node second-order triangle (3 nodes associated with the vertices and 3 with the edges).</td>
</tr>
<tr>
<td>QUAD4</td>
<td>4-node quadrilateral.</td>
</tr>
<tr>
<td>QUAD8</td>
<td>8-node second-order quadrilateral (4 nodes associated with the vertices and 4 with the edges).</td>
</tr>
<tr>
<td>TETR4</td>
<td>4-node tetrahedron.</td>
</tr>
<tr>
<td>TETR10</td>
<td>10-node second-order tetrahedron (4 nodes associated with the vertices and 6 with the edges).</td>
</tr>
<tr>
<td>PYRA5</td>
<td>5-node pyramid.</td>
</tr>
<tr>
<td>PYRA13</td>
<td>13-node second-order pyramid (5 nodes associated with the vertices and 8 with the edges).</td>
</tr>
<tr>
<td>PRIS6</td>
<td>6-node prism.</td>
</tr>
<tr>
<td>PRIS15</td>
<td>15-node second-order prism (6 nodes associated with the vertices and 9 with the edges).</td>
</tr>
<tr>
<td>HEXA8</td>
<td>8-node hexahedron.</td>
</tr>
<tr>
<td>HEXA20</td>
<td>20-node second-order hexahedron (8 nodes associated with the vertices and 12 with the edges).</td>
</tr>
</tbody>
</table>

Figure B.1: Representation of the 3-node triangular element.
Figure B.2: Representation of barycentric coordinates for a triangle in physical space.

The reference element can be used to represent the space $L_1$, $L_2$, $L_3$ as shown in Fig. B.3. The variables beneath the integral symbol are altered as shown in Eq. B.3.

$$\int_x \int_y f(x, y) \, dxdy = \int_\xi \int_\eta f(\xi, \eta) |J| \, d\xi d\eta, \quad (B.3)$$

where $J$ is the Jacobian matrix of geometrical transformation, as described in Eq. 3.59. By convention, we number the nodes of the reference and physical elements in the positive sense of rotation on the oriented surface, starting with a vertex node.

B.1.2 3-Node Linear Triangle

Fig. B.1 shows the 3-node linear triangle in reference space and physical space, respectively. Accordingly, the shape functions of the 3-node triangle and their derivatives are listed in Table B.2.
Table B.2: Shape functions for the 3-node triangle and their derivatives.

<table>
<thead>
<tr>
<th>Node i</th>
<th>$\phi_i$</th>
<th>$\partial \phi_i / \partial \xi$</th>
<th>$\partial \phi_i / \partial \eta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1 - \xi - \eta$</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>$\xi$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$\eta$</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

B.1.3 6-Node Curvilinear Triangle

Fig. B.4 shows the 6-node curvilinear triangle in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.3.

Figure B.4: Representation of the 6-node triangular element.
Table B.3: Shape functions for the 6-node triangle and their derivatives.

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>$\phi_i$</th>
<th>$\partial \phi_i / \partial \xi$</th>
<th>$\partial \phi_i / \partial \eta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$-\lambda(1-2\lambda)$</td>
<td>$1-4\lambda$</td>
<td>$1-4\lambda$</td>
</tr>
<tr>
<td>2</td>
<td>$-\xi(1-2\xi)$</td>
<td>$-1+4\xi$</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$-\eta(1-2\eta)$</td>
<td>0</td>
<td>$-1+4\eta$</td>
</tr>
<tr>
<td>4</td>
<td>$4\xi\lambda$</td>
<td>$4(\lambda-\xi)$</td>
<td>$-4\xi$</td>
</tr>
<tr>
<td>5</td>
<td>$4\xi\eta$</td>
<td>$4\eta$</td>
<td>$4\xi$</td>
</tr>
<tr>
<td>6</td>
<td>$4\eta\lambda$</td>
<td>$-4\eta$</td>
<td>$4(\lambda-\eta)$</td>
</tr>
</tbody>
</table>

Note: $\lambda = 1 - \xi - \eta$
B.2 Quadrilateral Elements (two dimensions)

B.2.1 Systems of Coordinates

For all quadrilateral elements, the reference element shown in Fig. B.5 is used, where the coordinates \((\xi, \eta)\) can be interpreted as curvilinear coordinates for the physical element. By convention, nodes are numbered in the positive sense of rotation on the oriented surface of the element, starting with a vertex node.

\[
\begin{align*}
1 & \quad \xi = 1 \\
2 & \quad \eta = 1 \\
3 & \quad \eta = -1 \\
4 & \quad \xi = -1 \\
\end{align*}
\]

(a) QUAD4 in reference space

(b) QUAD4 in physical space

Figure B.5: Representation of the 4-node quadrilateral element.

The variables beneath the integral symbol are altered as shown in Eq. B.4.

\[
\int_{x}^{y} f(x, y) \, dx \, dy = \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta) \left| J \right| \, d\xi \, d\eta.
\]  

(B.4)

B.2.2 4-Node Bilinear Quadrilateral

Fig. B.5 shows the 4-node bilinear quadrilateral in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.4.

B.2.3 8-Node Curvilinear Quadrilateral

Fig. B.6 shows the 8-node curvilinear quadrilateral in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.5.
Table B.4: Shape functions for the 4-node quadrilateral and their derivatives.

<table>
<thead>
<tr>
<th>Node i</th>
<th>( \phi_i )</th>
<th>( \partial \phi_i / \partial \xi )</th>
<th>( \partial \phi_i / \partial \eta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \frac{1}{4}(1 - \xi)(1 - \eta)(-\eta - \xi - 1) )</td>
<td>( \frac{1}{4}(1 - \eta)(2\xi + \eta) )</td>
<td>( \frac{1}{4}(1 - \xi)(\xi + 2\eta) )</td>
</tr>
<tr>
<td>2</td>
<td>( \frac{1}{4}(1 + \xi)(1 - \eta)(-\eta + \xi - 1) )</td>
<td>( \frac{1}{4}(1 - \eta)(2\xi - \eta) )</td>
<td>( \frac{1}{4}(1 + \xi)(2\eta - \xi) )</td>
</tr>
<tr>
<td>3</td>
<td>( \frac{1}{4}(1 + \xi)(1 + \eta)(\eta + \xi - 1) )</td>
<td>( \frac{1}{4}(1 + \eta)(2\xi + \eta) )</td>
<td>( \frac{1}{4}(1 + \xi)(\xi + 2\eta) )</td>
</tr>
<tr>
<td>4</td>
<td>( \frac{1}{4}(1 - \xi)(1 + \eta)(\eta - \xi - 1) )</td>
<td>( \frac{1}{4}(1 + \eta)(2\xi - \eta) )</td>
<td>( \frac{1}{4}(1 - \xi)(2\eta - \xi) )</td>
</tr>
</tbody>
</table>

(a) QUAD8 in reference space  
(b) QUAD8 in physical space

Figure B.6: Representation of the 8-node quadrilateral element.

Table B.5: Shape functions for the 8-node quadrilateral and their derivatives.

<table>
<thead>
<tr>
<th>Node i</th>
<th>( \phi_i )</th>
<th>( \partial \phi_i / \partial \xi )</th>
<th>( \partial \phi_i / \partial \eta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \frac{1}{8}(1 - \xi)(1 - \eta)(-\eta - \xi - 1) )</td>
<td>( \frac{1}{8}(1 - \eta)(2\xi + \eta) )</td>
<td>( \frac{1}{8}(1 - \xi)(\xi + 2\eta) )</td>
</tr>
<tr>
<td>2</td>
<td>( \frac{1}{8}(1 + \xi)(1 - \eta)(-\eta + \xi - 1) )</td>
<td>( \frac{1}{8}(1 - \eta)(2\xi - \eta) )</td>
<td>( \frac{1}{8}(1 + \xi)(2\eta - \xi) )</td>
</tr>
<tr>
<td>3</td>
<td>( \frac{1}{8}(1 + \xi)(1 + \eta)(\eta + \xi - 1) )</td>
<td>( \frac{1}{8}(1 + \eta)(2\xi + \eta) )</td>
<td>( \frac{1}{8}(1 + \xi)(\xi + 2\eta) )</td>
</tr>
<tr>
<td>4</td>
<td>( \frac{1}{8}(1 - \xi)(1 + \eta)(\eta - \xi - 1) )</td>
<td>( \frac{1}{8}(1 + \eta)(2\xi - \eta) )</td>
<td>( \frac{1}{8}(1 - \xi)(2\eta - \xi) )</td>
</tr>
<tr>
<td>5</td>
<td>( \frac{1}{8}(1 - \xi^2)(1 - \eta) )</td>
<td>( -1 - \eta \xi )</td>
<td>( -1 - \xi \eta )</td>
</tr>
<tr>
<td>6</td>
<td>( \frac{1}{8}(1 - \eta^2)(1 + \xi) )</td>
<td>( \frac{1}{2}(1 - \eta^2) )</td>
<td>( -\frac{1}{2}(1 - \xi^2) )</td>
</tr>
<tr>
<td>7</td>
<td>( \frac{1}{8}(1 - \xi^2)(1 + \eta) )</td>
<td>( -1 + \eta \xi )</td>
<td>( \frac{1}{2}(1 - \xi^2) )</td>
</tr>
<tr>
<td>8</td>
<td>( \frac{1}{8}(1 - \eta^2)(1 - \xi) )</td>
<td>( \frac{1}{2}(1 - \eta^2) )</td>
<td>( -1 - \xi \eta )</td>
</tr>
</tbody>
</table>

202
B.3 Tetrahedral Elements

B.3.1 Systems of Coordinates

For all tetrahedral-shaped elements, the reference element shown in Fig. B.7 is used.

![Tetrahedral Elements Diagram](image)

Figure B.7: Representation of the 4-node tetrahedral element.

Just like for triangles, the coordinates $(\xi, \eta, \zeta)$ can be interpreted as curvilinear coordinates for the physical element. The surface $\xi = \text{constant}$ (or $\eta = \text{constant}$ or $\zeta = \text{constant}$) is plane, parallel to the faces of the element, in the case of an element with straight edges.

The barycentric coordinates $L_1, L_2, L_3$ and $L_4$ are sometimes used to represent a point $0$ for a straight-edged tetrahedron as show in Fig. B.8.

\[
L_1 = \frac{A_1}{A}, \quad L_2 = \frac{A_2}{A}, \quad L_3 = \frac{A_3}{A}, \quad L_4 = \frac{A_4}{A},
\]

\[
A = A_1 + A_2 + A_3 + A_4,
\]

\[
L_1 + L_2 + L_3 + L_4 = 1,
\]

where $A_1, A_2, A_3$ and $A_4$ are the volume of the tetrahedra 0-2-3-4, 0-1-3-4, 0-1-2-4 and 0-1-2-3, respectively. $A$ is the volume of the whole tetrahedron. The barycentric coordinates are linked to the coordinates $(\xi, \eta, \zeta)$ by the following relation:

\[
L_1 \equiv 1 - \xi - \eta - \zeta, \quad L_2 \equiv \xi, \quad L_3 \equiv \eta, \quad L_4 \equiv \zeta.
\]
Figure B.8: Representation of barycentric coordinates for a tetrahedron in physical space.

Note that the order of the numbering must be consistent between the reference element and the physical element. Here the first three nodes are taken in the positive sense of rotation about a unit normally oriented toward the interior of the element.

The variables beneath the integral symbol are altered as follows:

$$\int_x \int_y \int_z f(x, y, z) \, dxdydz = \int_0^1 \int_0^{1-\zeta} \int_0^{1-\eta-\zeta} f(\xi, \eta, \zeta) |J| \, d\xi d\eta d\zeta.$$  \hfill (B.7)

### B.3.2 4-node Linear Tetrahedron

Fig. B.7 shows the 4-node linear tetrahedron in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.6.

Table B.6: Shape functions for the 4-node tetrahedron and their derivatives.

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>$\phi_i$</th>
<th>$\partial \phi_i / \partial \xi$</th>
<th>$\partial \phi_i / \partial \eta$</th>
<th>$\partial \phi_i / \partial \zeta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1 - \xi - \eta - \zeta$</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>$\xi$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$\eta$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>$\zeta$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
B.3.3 10-node Curvilinear Tetrahedron

Fig. B.9 shows the 10-node curvilinear tetrahedron in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.7.

Figure B.9: Representation of the 10-node tetrahedral element.
Table B.7: Shape functions for the 10-node tetrahedron and their derivatives.

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>$\phi_i$</th>
<th>$\partial \phi_i / \partial \xi$</th>
<th>$\partial \phi_i / \partial \eta$</th>
<th>$\partial \phi_i / \partial \zeta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$-\lambda(1 - 2\lambda)$</td>
<td>$1 - 4\lambda$</td>
<td>$1 - 4\lambda$</td>
<td>$1 - 4\lambda$</td>
</tr>
<tr>
<td>2</td>
<td>$-\xi(1 - 2\xi)$</td>
<td>$-1 + 4\xi$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$-\eta(1 - 2\eta)$</td>
<td>0</td>
<td>$-1 + 4\eta$</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>$-\zeta(1 - 2\zeta)$</td>
<td>0</td>
<td>0</td>
<td>$-1 + 4\zeta$</td>
</tr>
<tr>
<td>5</td>
<td>$4\xi \lambda$</td>
<td>$4(\lambda - \xi)$</td>
<td>$-4\xi$</td>
<td>$-4\xi$</td>
</tr>
<tr>
<td>6</td>
<td>$4\xi \eta$</td>
<td>$4\eta$</td>
<td>$4\xi$</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>$4\eta \lambda$</td>
<td>$-4\eta$</td>
<td>$4(\lambda - \eta)$</td>
<td>$-4\eta$</td>
</tr>
<tr>
<td>8</td>
<td>$4\xi \zeta$</td>
<td>$-4\xi$</td>
<td>$-4\zeta$</td>
<td>$4(\lambda - \zeta)$</td>
</tr>
<tr>
<td>9</td>
<td>$4\xi \zeta$</td>
<td>$4\xi$</td>
<td>0</td>
<td>$4\xi$</td>
</tr>
<tr>
<td>10</td>
<td>$4\eta \zeta$</td>
<td>0</td>
<td>$4\zeta$</td>
<td>$4\eta$</td>
</tr>
</tbody>
</table>

Note: $\lambda = 1 - \xi - \eta - \zeta$
B.4 Hexahedral Elements

B.4.1 8-Node Trilinear Hexahedron

Fig. B.10 shows the 8-node trilinear hexahedron in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.8.

Figure B.10: Representation of the 8-node hexahedral element.

Table B.8: Shape functions for the 8-node hexahedron and their derivatives.

<table>
<thead>
<tr>
<th>Node i</th>
<th>( \phi_i )</th>
<th>( \partial \phi_i / \partial \xi )</th>
<th>( \partial \phi_i / \partial \eta )</th>
<th>( \partial \phi_i / \partial \zeta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \frac{1}{8}(1 - \xi)(1 - \eta)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 - \eta)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 - \xi)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 - \xi)(1 - \eta) )</td>
</tr>
<tr>
<td>2</td>
<td>( \frac{1}{8}(1 + \xi)(1 - \eta)(1 - \zeta) )</td>
<td>( \frac{1}{8}(1 - \eta)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 + \xi)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 + \xi)(1 - \eta) )</td>
</tr>
<tr>
<td>3</td>
<td>( \frac{1}{8}(1 + \xi)(1 + \eta)(1 - \zeta) )</td>
<td>( \frac{1}{8}(1 + \eta)(1 - \zeta) )</td>
<td>( \frac{1}{8}(1 + \xi)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 + \xi)(1 + \eta) )</td>
</tr>
<tr>
<td>4</td>
<td>( \frac{1}{8}(1 - \xi)(1 + \eta)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 + \eta)(1 - \zeta) )</td>
<td>( \frac{1}{8}(1 - \xi)(1 - \zeta) )</td>
<td>( -\frac{1}{8}(1 - \xi)(1 + \eta) )</td>
</tr>
<tr>
<td>5</td>
<td>( \frac{1}{8}(1 - \xi)(1 - \eta)(1 + \zeta) )</td>
<td>( -\frac{1}{8}(1 - \eta)(1 + \zeta) )</td>
<td>( -\frac{1}{8}(1 - \xi)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 - \xi)(1 + \eta) )</td>
</tr>
<tr>
<td>6</td>
<td>( \frac{1}{8}(1 + \xi)(1 - \eta)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 - \eta)(1 + \zeta) )</td>
<td>( -\frac{1}{8}(1 + \xi)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 + \xi)(1 - \eta) )</td>
</tr>
<tr>
<td>7</td>
<td>( \frac{1}{8}(1 + \xi)(1 + \eta)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 + \eta)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 + \xi)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 + \xi)(1 + \eta) )</td>
</tr>
<tr>
<td>8</td>
<td>( \frac{1}{8}(1 - \xi)(1 + \eta)(1 + \zeta) )</td>
<td>( -\frac{1}{8}(1 + \eta)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 - \xi)(1 + \zeta) )</td>
<td>( \frac{1}{8}(1 - \xi)(1 + \eta) )</td>
</tr>
</tbody>
</table>
B.4.2 20-Node Curvilinear Hexahedron

Fig. B.11 shows the 20-node curvilinear hexahedron in reference space and physical space, respectively.

![20-node hexahedral element](image)

Figure B.11: Representation of the 20-node hexahedral element.

The shape functions and their derivatives are presented as follows:

— Vertex nodes:

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi_i$</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>$\eta_i$</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>$\zeta_i$</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\[ \phi_i = \frac{1}{8} (1 + \xi_i)(1 + \eta_i)(1 + \zeta_i)(-2 + \xi_i + \eta_i + \zeta_i), \]
\[ \frac{\partial \phi_i}{\partial \xi} = \frac{1}{8} \xi_i(1 + \eta_i)(1 + \zeta_i)(-1 + 2\xi_i + \eta_i + \zeta_i), \]
\[ \frac{\partial \phi_i}{\partial \eta} = \frac{1}{8} \eta_i(1 + \xi_i)(1 + \zeta_i)(-1 + \xi_i + 2\eta_i + \zeta_i), \]
\[ \frac{\partial \phi_i}{\partial \zeta} = \frac{1}{8} \zeta_i(1 + \xi_i)(1 + \eta_i)(-1 + \xi_i + \eta_i + 2\zeta_i). \]  \hspace{1cm} (B.8)

— Nodes on the edges parallel to the \( \xi \) axis:

<table>
<thead>
<tr>
<th>Node ( i )</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \xi_i )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( \eta_i )</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>( \zeta_i )</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \phi_i = \frac{1}{4} (1 + \xi^2)(1 + \eta_i)(1 + \zeta_i), \]
\[ \frac{\partial \phi_i}{\partial \xi} = -\frac{1}{2} \xi(1 + \eta_i)(1 + \zeta_i), \]
\[ \frac{\partial \phi_i}{\partial \eta} = \frac{1}{4} \eta_i(1 - \xi^2)(1 + \zeta_i), \]
\[ \frac{\partial \phi_i}{\partial \zeta} = \frac{1}{4} \zeta_i(1 - \xi^2)(1 + \eta_i). \]  \hspace{1cm} (B.9)

— Nodes on the edges parallel to the \( \eta \) axis:

<table>
<thead>
<tr>
<th>Node ( i )</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \xi_i )</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>( \eta_i )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( \zeta_i )</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\[
\phi_i = \frac{1}{4}(1 + \xi \xi_i)(1 - \eta^2)(1 + \zeta \zeta_i),
\]
\[
\frac{\partial \phi_i}{\partial \xi} = \frac{1}{4} \xi_i(1 - \eta^2)(1 + \zeta \zeta_i),
\]
\[
\frac{\partial \phi_i}{\partial \eta} = -\frac{1}{2} \eta(1 + \xi \xi_i)(1 + \zeta \zeta_i),
\]
\[
\frac{\partial \phi_i}{\partial \zeta} = \frac{1}{4} \zeta_i(1 + \xi \xi_i)(1 - \eta^2).
\] (B.10)

— Nodes on the edges parallel to the \( \zeta \) axis:

<table>
<thead>
<tr>
<th>Node ( i )</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \xi )</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>( \eta )</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \zeta )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\phi_i = \frac{1}{4}(1 + \xi \xi_i)(1 + \eta \eta_i)(1 - \zeta^2),
\]
\[
\frac{\partial \phi_i}{\partial \xi} = \frac{1}{4} \xi_i(1 + \eta \eta_i)(1 - \zeta^2),
\]
\[
\frac{\partial \phi_i}{\partial \eta} = \frac{1}{4} \eta_i(1 + \xi \xi_i)(1 + \zeta^2),
\]
\[
\frac{\partial \phi_i}{\partial \zeta} = -\frac{1}{2} \zeta_i(1 + \xi \xi_i)(1 + \eta \eta_i).
\] (B.11)
B.5 Prismatic Elements

B.5.1 6-Node Prism

Fig. B.12 shows the 6-node prism in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.9.

![Figure B.12: Representation of the 6-node prismatic element.](image)

Table B.9: Shape functions for the 6-node prism and their derivatives.

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>$\phi_i$</th>
<th>$\partial\phi_i/\partial\xi$</th>
<th>$\partial\phi_i/\partial\eta$</th>
<th>$\partial\phi_i/\partial\zeta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\lambda a$</td>
<td>$-a$</td>
<td>$-a$</td>
<td>$-\frac{1}{2}\lambda$</td>
</tr>
<tr>
<td>2</td>
<td>$\xi a$</td>
<td>$a$</td>
<td>$0$</td>
<td>$-\frac{1}{2}\zeta$</td>
</tr>
<tr>
<td>3</td>
<td>$\eta a$</td>
<td>$0$</td>
<td>$a$</td>
<td>$-\frac{1}{2}\eta$</td>
</tr>
<tr>
<td>4</td>
<td>$\lambda b$</td>
<td>$-b$</td>
<td>$-b$</td>
<td>$\frac{1}{2}\lambda$</td>
</tr>
<tr>
<td>5</td>
<td>$\xi b$</td>
<td>$b$</td>
<td>$0$</td>
<td>$\frac{1}{2}\zeta$</td>
</tr>
<tr>
<td>6</td>
<td>$\eta b$</td>
<td>$0$</td>
<td>$b$</td>
<td>$\frac{1}{2}\eta$</td>
</tr>
</tbody>
</table>

Note: $\lambda = 1 - \xi - \eta$, $a = \frac{1}{2}(1 - \zeta)$, $b = \frac{1}{2}(1 + \zeta)$
B.5.2 15-node Prism

Fig. B.13 shows the 15-node prism in reference space and physical space, respectively. Accordingly, the shape functions and their derivatives are listed in Table B.10.

Figure B.13: Representation of the 15-node prismatic element.
Table B.10: Shape functions for the 15-node prism and their derivatives.

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>$\phi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a\lambda(2\lambda - \zeta - 2)$</td>
</tr>
<tr>
<td>2</td>
<td>$a\xi(2\xi - \zeta - 2)$</td>
</tr>
<tr>
<td>3</td>
<td>$a\eta(2\eta - \zeta - 2)$</td>
</tr>
<tr>
<td>4</td>
<td>$b\lambda(2\lambda + \zeta - 2)$</td>
</tr>
<tr>
<td>5</td>
<td>$b\xi(2\xi + \zeta - 2)$</td>
</tr>
<tr>
<td>6</td>
<td>$b\eta(2\eta + \zeta - 2)$</td>
</tr>
<tr>
<td>7</td>
<td>$4a\lambda\xi$</td>
</tr>
<tr>
<td>8</td>
<td>$4a\eta\xi$</td>
</tr>
<tr>
<td>9</td>
<td>$4a\eta\lambda$</td>
</tr>
<tr>
<td>10</td>
<td>$4b\lambda\xi$</td>
</tr>
<tr>
<td>11</td>
<td>$4b\eta\xi$</td>
</tr>
<tr>
<td>12</td>
<td>$4b\eta\lambda$</td>
</tr>
<tr>
<td>13</td>
<td>$(1 - \zeta^2)\lambda$</td>
</tr>
<tr>
<td>14</td>
<td>$(1 - \zeta^2)\xi$</td>
</tr>
<tr>
<td>15</td>
<td>$(1 - \zeta^2)\eta$</td>
</tr>
</tbody>
</table>

Note: $\lambda = 1 - \xi - \eta - \zeta$, $a = \frac{1}{2}(1 - \xi)$, $b = \frac{1}{2}(1 + \xi)$
B.6 Pyramidal Elements

B.6.1 5-Node Pyramid

Fig. B.14 shows the 5-node pyramid in reference space and physical space, respectively.

![Pyramid in reference space](image1)

![Pyramid in physical space](image2)

Figure B.14: Representation of the 5-node pyramidal element.

The shape functions of 5-node pyramid and their derivatives are listed in Table B.11

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>$\phi_i$</th>
<th>$\partial \phi_i / \partial \xi$</th>
<th>$\partial \phi_i / \partial \eta$</th>
<th>$\partial \phi_i / \partial \zeta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\frac{1}{8}(1 - \xi)(1 - \eta)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 - \eta)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 - \xi)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 - \xi)(1 - \eta)$</td>
</tr>
<tr>
<td>2</td>
<td>$\frac{1}{8}(1 + \xi)(1 - \eta)(1 - \zeta)$</td>
<td>$\frac{1}{8}(1 - \eta)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 + \xi)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 + \xi)(1 - \eta)$</td>
</tr>
<tr>
<td>3</td>
<td>$\frac{1}{8}(1 + \xi)(1 + \eta)(1 - \zeta)$</td>
<td>$\frac{1}{8}(1 + \eta)(1 - \zeta)$</td>
<td>$\frac{1}{8}(1 + \xi)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 + \xi)(1 + \eta)$</td>
</tr>
<tr>
<td>4</td>
<td>$\frac{1}{8}(1 - \xi)(1 + \eta)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 + \eta)(1 - \zeta)$</td>
<td>$\frac{1}{8}(1 - \xi)(1 - \zeta)$</td>
<td>$-\frac{1}{8}(1 - \xi)(1 + \eta)$</td>
</tr>
<tr>
<td>5</td>
<td>$\frac{1}{2}(1 + \zeta)$</td>
<td>0</td>
<td>0</td>
<td>$\frac{1}{2}$</td>
</tr>
</tbody>
</table>
B.6.2 13-Node Pyramid

Fig. B.15 shows the 13-node pyramid in reference space and physical space, respectively. Accordingly, the shape functions of 13-node pyramid are listed in Table B.12.

![13-node pyramid in reference space](image1)

![13-node pyramid in physical space](image2)

Figure B.15: Representation of the 13-node pyramidal element.
Table B.12: Shape functions for the 13-node pyramid.

<table>
<thead>
<tr>
<th>Node $i$</th>
<th>$\phi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\frac{1}{8}(1 - \xi)(1 - \eta)(1 - \zeta)(-2 - \xi - \eta - \zeta)$</td>
</tr>
<tr>
<td>2</td>
<td>$\frac{1}{8}(1 + \xi)(1 - \eta)(1 - \zeta)(-2 + \xi - \eta - \zeta)$</td>
</tr>
<tr>
<td>3</td>
<td>$\frac{1}{8}(1 + \xi)(1 + \eta)(1 - \zeta)(-2 + \xi + \eta - \zeta)$</td>
</tr>
<tr>
<td>4</td>
<td>$\frac{1}{8}(1 - \xi)(1 + \eta)(1 - \zeta)(-2 - \xi + \eta - \zeta)$</td>
</tr>
<tr>
<td>5</td>
<td>$\frac{1}{2}\zeta(1 + \zeta)$</td>
</tr>
<tr>
<td>6</td>
<td>$\frac{1}{8}(1 - \xi^2)(1 - \eta)(1 - \zeta)$</td>
</tr>
<tr>
<td>7</td>
<td>$\frac{1}{8}(1 + \xi)(1 - \eta^2)(1 - \zeta)$</td>
</tr>
<tr>
<td>8</td>
<td>$\frac{1}{8}(1 + \xi^2)(1 + \eta)(1 - \zeta)$</td>
</tr>
<tr>
<td>9</td>
<td>$\frac{1}{8}(1 - \xi)(1 + \eta^2)(1 - \zeta)$</td>
</tr>
<tr>
<td>10</td>
<td>$\frac{1}{8}(1 - \xi^2)(1 - \eta)(1 + \zeta)$</td>
</tr>
<tr>
<td>11</td>
<td>$\frac{1}{8}(1 + \xi)(1 - \eta^2)(1 + \zeta)$</td>
</tr>
<tr>
<td>12</td>
<td>$\frac{1}{8}(1 + \xi^2)(1 + \eta)(1 + \zeta)$</td>
</tr>
<tr>
<td>13</td>
<td>$\frac{1}{8}(1 - \xi)(1 + \eta^2)(1 + \zeta)$</td>
</tr>
</tbody>
</table>
In this section all the used Gaussian quadrature points and weighting coefficients for various types of two-dimensional and three-dimensional elements are presented:

- §C.1 gives the integration constants of Gaussian quadrature rules for a triangle.
- §C.2 gives the integration constants of Gaussian quadrature rules for a quadrilateral.
- §C.3 gives the integration constants of Gaussian quadrature rules for a tetrahedron.
- §C.4 gives the integration constants of Gaussian quadrature rules for a hexahedron.
- §C.5 gives the integration constants of Gaussian quadrature rules for a prism.
- §C.6 gives the integration constants of Gaussian quadrature rules for a pyramid.

More detailed description is referred to Hesthaven [58], Dhatt [36] and Li [72].
### C.1 Triangle

Table C.1: Integration constants of Gaussian quadrature rules for a triangle.

<table>
<thead>
<tr>
<th>Order $p$</th>
<th>No. of point $r$</th>
<th>Coordinates $(\xi, \eta)$</th>
<th>Weights $w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(0.333333333333333, 0.333333333333333)</td>
<td>0.500000000000000</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>(0.166666666666667, 0.666666666666667)</td>
<td>0.166666666666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.666666666666667, 0.166666666666667)</td>
<td>0.166666666666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.166666666666667, 0.666666666666667)</td>
<td>0.166666666666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.666666666666667, 0.166666666666667)</td>
<td>0.166666666666667</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>(0.333333333333333, 0.333333333333333)</td>
<td>-0.281250000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.200000000000000, 0.200000000000000)</td>
<td>0.260416666666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.600000000000000, 0.200000000000000)</td>
<td>0.260416666666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.200000000000000, 0.600000000000000)</td>
<td>0.260416666666667</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>(0.333333333333333, 0.333333333333333)</td>
<td>0.112500000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.059715871789770, 0.470142064105115)</td>
<td>0.066197063942530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.470142064105115, 0.059715871789770)</td>
<td>0.066197063942530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.470142064105115, 0.470142064105115)</td>
<td>0.066197063942530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.797426985353087, 0.101286507323456)</td>
<td>0.062969590272414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.101286507323456, 0.797426985353087)</td>
<td>0.062969590272414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.101286507323456, 0.101286507323456)</td>
<td>0.062969590272414</td>
</tr>
</tbody>
</table>

Formulas precisely integrating the $m$-th order polynomials:

$$
\int_0^1 \int_0^{1-\xi} f(\xi, \eta) \, d\xi d\eta \simeq \sum_i^r w_i f(\xi_i, \eta_i). \tag{C.1}
$$
### C.2 Quadrilateral

Table C.2: Integration constants of Gaussian quadrature rules for a quadrilateral.

<table>
<thead>
<tr>
<th>Order $p$</th>
<th>No. of points $r$</th>
<th>Coordinates $(\xi_i, \eta_i)$</th>
<th>Weights $w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$(0.000000000000000, 0.000000000000000)$</td>
<td>1.00000000000000</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$(0.577350269189625, -0.577350269189625, 0.577350269189625, -0.577350269189625)$</td>
<td>1.00000000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.577350269189625, -0.577350269189625, 0.577350269189625, -0.577350269189625)$</td>
<td>1.00000000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.577350269189625, -0.577350269189625, 0.577350269189625, -0.577350269189625)$</td>
<td>1.00000000000000</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>$(0.774596669241483, -0.774596669241483, 0.000000000000000, 0.000000000000000)$</td>
<td>0.308641975308642</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.774596669241483, -0.774596669241483, 0.000000000000000, 0.000000000000000)$</td>
<td>0.493827160493827</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.774596669241483, -0.774596669241483, 0.000000000000000, 0.000000000000000)$</td>
<td>0.308641975308642</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.774596669241483, -0.774596669241483, 0.000000000000000, 0.000000000000000)$</td>
<td>0.493827160493827</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.774596669241483, -0.774596669241483, 0.000000000000000, 0.000000000000000)$</td>
<td>0.308641975308642</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.774596669241483, -0.774596669241483, 0.000000000000000, 0.000000000000000)$</td>
<td>0.493827160493827</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.774596669241483, -0.774596669241483, 0.000000000000000, 0.000000000000000)$</td>
<td>0.308641975308642</td>
</tr>
</tbody>
</table>

Formulas integrating exactly the polynomials of order $m$:

$$
\int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta) \, d\xi d\eta \approx \sum_{i}^{r} w_i f(\xi_i, \eta_i).
$$

(C.2)
## C.3 Tetrahedron

Table C.3: Integration constants of Gaussian quadrature rules for a tetrahedron.

<table>
<thead>
<tr>
<th>Order ( p )</th>
<th>No. of points ( r )</th>
<th>Coordinates ((\xi_i, \eta_i, \zeta_i))</th>
<th>Weights ( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>((0.250000000000000, 0.250000000000000, 0.250000000000000))</td>
<td>(0.166666666666667)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>((0.138196601125011, 0.138196601125011, 0.138196601125011))</td>
<td>(0.041666666666667)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.585410196624969, 0.138196601125011, 0.138196601125011))</td>
<td>(0.041666666666667)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.138196601125011, 0.585410196624969, 0.138196601125011))</td>
<td>(0.041666666666667)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.138196601125011, 0.138196601125011, 0.585410196624969))</td>
<td>(0.041666666666667)</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>((0.250000000000000, 0.250000000000000, 0.250000000000000))</td>
<td>(-0.133333333333333)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.166666666666667, 0.166666666666667, 0.166666666666667))</td>
<td>(0.075000000000000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.500000000000000, 0.166666666666667, 0.166666666666667))</td>
<td>(0.075000000000000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.166666666666667, 0.500000000000000, 0.166666666666667))</td>
<td>(0.075000000000000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0.166666666666667, 0.166666666666667, 0.500000000000000))</td>
<td>(0.075000000000000)</td>
</tr>
</tbody>
</table>

Formulas integrating exactly the polynomials of order \( m \):

\[
\int_0^1 \int_0^{1-\xi} \int_0^{1-\xi-\eta} f(\xi, \eta, \zeta) \, d\xi d\eta d\zeta \simeq \sum_i^r w_i f(\xi_i, \eta_i, \zeta_i). \tag{C.3}
\]
C.4 Hexahedron

Table C.4: Integration constants of Gaussian quadrature rules for a hexahedron.

<table>
<thead>
<tr>
<th>Order $p$</th>
<th>No. of points $r$</th>
<th>Coordinates $(\xi, \eta, \zeta)$</th>
<th>Weights $w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$r$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.00000000000000000000, 0.00000000000000000000, 0.00000000000000000000)$</td>
<td>$8.00000000000000000000$</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>$r$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(-0.577350269189625, -0.577350269189625, -0.577350269189625, 0.577350269189625, 0.577350269189625)$</td>
<td>$1.00000000000000000000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(-0.577350269189625, -0.577350269189625, 0.577350269189625, -0.577350269189625, 0.577350269189625)$</td>
<td>$1.00000000000000000000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(-0.577350269189625, 0.577350269189625, -0.577350269189625, -0.577350269189625, 0.577350269189625)$</td>
<td>$1.00000000000000000000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.577350269189625, -0.577350269189625, -0.577350269189625, 0.577350269189625, 0.577350269189625)$</td>
<td>$1.00000000000000000000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.577350269189625, -0.577350269189625, 0.577350269189625, -0.577350269189625, 0.577350269189625)$</td>
<td>$1.00000000000000000000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(0.577350269189625, 0.577350269189625, -0.577350269189625, -0.577350269189625, 0.577350269189625)$</td>
<td>$1.00000000000000000000$</td>
</tr>
</tbody>
</table>

Formulas precisely integrating the $m$-th order polynomials:

$$
\int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta, \zeta) \ d\xi d\eta d\zeta \simeq \sum_{i}^{r} w_i f(\xi_i, \eta_i, \zeta_i). \quad (C.4)
$$
### C.5 Prism

Table C.5: Integration constants of Gaussian quadrature rules for a prism.

<table>
<thead>
<tr>
<th>Order ( p )</th>
<th>No. of points ( r )</th>
<th>Coordinates ((\xi_i, \eta_i, \zeta_i))</th>
<th>Weights ( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( { 0.333333333333333, 0.333333333333333, 0.000000000000000 } )</td>
<td>( 1.000000000000000 )</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>( { 0.166666666666667, 0.166666666666667, -0.577350269189625 } )</td>
<td>( 0.166666666666667 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { 0.166666666666667, 0.166666666666667, -0.577350269189625 } )</td>
<td>( 0.166666666666667 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { 0.166666666666667, 0.166666666666667, -0.577350269189625 } )</td>
<td>( 0.166666666666667 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { 0.166666666666667, 0.166666666666667, -0.577350269189625 } )</td>
<td>( 0.166666666666667 )</td>
</tr>
</tbody>
</table>

Formulas precisely integrating the \( m \)-th order polynomials:

\[
\int_0^1 \int_0^{1-\xi} \int_{-1}^1 f(\xi, \eta, \zeta) \, d\xi d\eta d\zeta \simeq \sum_i w_i f(\xi_i, \eta_i, \zeta_i). \quad (C.5)
\]
C.6 Pyramid

Table C.6: Integration constants of Gaussian quadrature rules for a pyramid.

<table>
<thead>
<tr>
<th>Order p</th>
<th>No. of points r</th>
<th>Coordinates $(\xi, \eta, \zeta)$</th>
<th>Weights $w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(0.000000000000000, 0.000000000000000, −0.500000000000000)</td>
<td>4.740740740740741</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>(−0.584237394672177, −0.584237394672177, −0.666666666666667)</td>
<td>0.810000000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.584237394672177, −0.584237394672177, −0.666666666666667)</td>
<td>0.810000000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(−0.584237394672177, 0.584237394672177, −0.666666666666667)</td>
<td>0.810000000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.000000000000000, 0.000000000000000, 0.400000000000000)</td>
<td>4.629629629629630</td>
</tr>
</tbody>
</table>

Formulas precisely integrating the $m$-th order polynomials:

$$\int_{\xi} \int_{\eta} \int_{\zeta} f(\xi, \eta, \zeta) \, d\xi \, d\eta \, d\zeta \simeq \sum_{i} w_i f(\xi_i, \eta_i, \zeta_i).$$

(C.6)