RocfluMP
Developer and Reference Manual

A. Haselbacher
Principal Research Scientist
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign
2253 Digital Computer Laboratory, MC-278
1304 West Springfield Avenue
Urbana, IL 61801

June 13, 2005

Contact information:
Andreas Haselbacher
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign
2253 Digital Computer Laboratory, MC-278
1304 West Springfield Avenue
Urbana, IL 61801
Phone: (217) 333-8683
Fax: (217) 333-8497
Email: haselbac@uiuc.edu

User's guide version: 1.2.0 of 03/17/05

This manual documents:
rfluconv version 2.10.0 of 05/16/05
rfluinit version 0.13.0 of 05/18/05
rflumap version 2.10.0 of 05/16/05
rflump version 5.13.0 of 05/19/05
rflupart version 0.11.1 of 05/16/05
rflupick version 4.9.0 of 05/16/05
rflupost version 6.13.0 of 05/18/05

Contents

1 Introduction **7**
- 1.1 Objective . . . 7
- 1.2 Overview of RocfluMP . . . 7
- 1.3 Overview of Developer's Guide . . . 10
- 1.4 Related Documents . . . 10

2 Nomenclature, Notation, and Conventions **11**
- 2.1 Nomenclature . . . 11
- 2.2 Notation . . . 11
- 2.3 Conventions . . . 12

3 Governing Equations **13**
- 3.1 The Navier-Stokes Equations . . . 13
- 3.2 The Geometric Conservation Law . . . 14
- 3.3 Gas Models . . . 15
  - 3.3.1 Calorically Perfect Gas . . . 15
  - 3.3.2 Thermally Perfect Gas . . . 15
- 3.4 Thermodynamic Properties . . . 15
- 3.5 Transport Properties . . . 15
  - 3.5.1 Viscosity . . . 15
  - 3.5.2 Conductivity . . . 15
- 3.6 Boundary Conditions . . . 15

4 Algorithms and Methods **17**
- 4.1 Geometry Definition . . . 17
  - 4.1.1 Computation of Face Properties . . . 17
  - 4.1.2 Computation of Volume Properties . . . 18
- 4.2 Spatial Discretization . . . 18
  - 4.2.1 Stencil Construction . . . 18
  - 4.2.2 Interpolation Operators . . . 19
  - 4.2.3 Gradient Operators . . . 20
  - 4.2.4 Inviscid Fluxes . . . 22
    - 4.2.4.1 Limiter Functions . . . 22

4.2.4.2 Numerical Flux Functions . . . 22
4.2.4.3 Entropy Fixes . . . 22
4.2.5 Viscous Fluxes . . . 22
4.2.6 Optimal LES Discretization . . . 22
4.2.6.1 Computation of Integrals . . . 22
4.2.6.2 Computation of Stencil Weights . . . 22
4.2.7 Source Terms . . . 22
4.3 Boundary Conditions . . . 22
4.3.1 Fluid-Solid Boundary Conditions . . . 22
4.3.1.1 Slip Wall Boundaries . . . 22
4.3.1.2 No-Slip Wall Boundaries . . . 22
4.3.1.3 Injection Wall Boundaries . . . 22
4.3.2 Fluid-Fluid Boundary Conditions . . . 22
4.3.2.1 Inflow Boundaries . . . 22
4.3.2.2 Outflow Boundaries . . . 22
4.3.2.3 Periodic Boundaries . . . 22
4.3.2.4 Symmetry Boundaries . . . 22
4.4 Temporal Discretization . . . 22
4.4.1 Runge-Kutta Methods . . . 22
4.4.2 Computation of Time Step . . . 22
4.5 Grid Motion . . . 22
4.5.1 Grid Smoothing . . . 22
4.5.2 Discrete Geometric Conservation Law . . . 22
4.5.3 Implementation Details . . . 22
4.6 Pressure, Skin-Friction, and Heat-Transfer Coefficient Computation . . . 22
4.7 Force and Moment Computation . . . 23

5 Code Design and Organization 27
5.1 Directory Structure . . . 27
5.2 Control Flow . . . 29
5.2.1 Standalone Code . . . 29
5.2.2 Code Coupled With GENx . . . 31
5.3 Error Handling . . . 31

6 Data Structures 33
6.1 Philosophy and Abstraction . . . 33
6.2 Region Data Structure . . . 34
6.3 Grid Data Structure . . . 36
6.3.1 Module ModGrid.F90 . . . 36
6.3.2 Module RFLU_ModGrid.F90 . . . 44
6.4 Boundary Data Structure . . . 44
6.5 Mixture Data Structure . . . 49

6.5.1 Data Type `t_mixt_input` 49
6.5.2 Data Type `t_mixt` 51

7 Parallel Implementation **55**

8 GENx Integration **57**

9 Installation and Compilation **59**
9.1 Installation 59
9.1.1 Installation from CVS Repository 59
9.1.2 Installation from `.tar.gz` File 60
9.2 Compilation 60
9.2.1 Overview of Compilation Process 60
9.2.2 Description of Compilation Options 61

References **63**

Chapter 1

Introduction

1.1 Objective

The objective of this developer's guide is two-fold:

1. To enable people other than the main developer(s) to modify and extend the RocfluMP source code by providing detailed technical information.

2. To enable people other than the main developer(s) to compile and install the RocfluMP source code on new computer systems.

1.2 Overview of RocfluMP

RocfluMP solves the three-dimensional time-dependent compressible Navier-Stokes equations on moving and/or deforming unstructured grids. The grids may consist of arbitrary combinations of tetrahedra, hexahedra, prisms, and pyramids. The spatial discretization is carried out using the finite-volume method. The inviscid fluxes are approximated by upwind schemes to allow for capturing of shock waves and contact discontinuities. Steady flows can be computed using an explicit multi-stage method tuned for fast convergence. Unsteady flows are computed with the fourth-order accurate Runge-Kutta method.

To solve fluid-dynamics problems in which processes other than those described by the Navier-Stokes equations are important, RocfluMP is designed to interface with so-called *multi-physics modules*. The multi-physics modules model phenomena such as turbulence, particles, chemical reactions, and radiation and their interaction. At present, the multi-physics modules are under development and have not yet been integrated with RocfluMP. When considering fluid flow problems with several chemical species, RocfluMP may be regarded as solving transport equations for the mixture variables.

RocfluMP may be used to solve problems involving fluid-structure interactions. More specifically, RocfluMP is designed to operate as a solution module inside CSAR's coupled rocket-simulation code GENx. To accommodate dynamically changing fluid domains arising

from the deformation predicted by a structural simulation, RocfluMP allows for moving grids. The Geometric Conservation Law (GCL) is satisfied in a discrete sense to machine-precision to avoid the introduction of spurious sources of mass, momentum, and energy due to grid motion.

The relationship of RocfluMP and the other codes is depicted schematically in Fig. 1.1. A brief description of the multi-physics modules follows (they are described in detail in their respective manuals).

- Rocturb is the turbulence module which implements a variety of models for Reynolds-averaged Navier-Stokes (RANS) simulations and Large-Eddy Simulation (LES).
- Rocpart is the Lagrangian particle tracking module.
- Rocspecies is the module simulating the evolution of chemical species and Equilibrium Eulerian particles.
- Rocrad is the radiation module.

RocfluMP consists of several modules:

- rfluconv is the conversion module of RocfluMP. It converts RocfluMP solution and grid files from ASCII to binary format and vice versa, and converts RocfluMP grid files into a format supported by YAMS and TETMESH. rfluconv requires interactive user input.
- rfluinit is the initialization module of RocfluMP. It generates RocfluMP solution files for each region based on the information contained in the user input file.
- rflumap is the processor-mapping module of RocfluMP. It generates the mapping file which is required for parallel computations. It also generates the Rocin control files for computations with GENx. rflumap requires interactive user input.
- rflump is the actual solution module of RocfluMP.
- rflupick is used in conjunction with rflupost to visualize only specific cells in the grid, such as cells near boundaries or cells sharing faces or vertices. For parallel computations, rflupick can also be used to instruct rflupost to convert only specific regions for visualization. This allows the visualization of nominally large cases on small machines. rflupick requires interactive user input.
- rflupost is the postprocessing module of RocfluMP. It converts grid and solution files from the RocfluMP format into the formats recognized by visualization packages. At present, the following formats are supported:
  - TECPLOT format

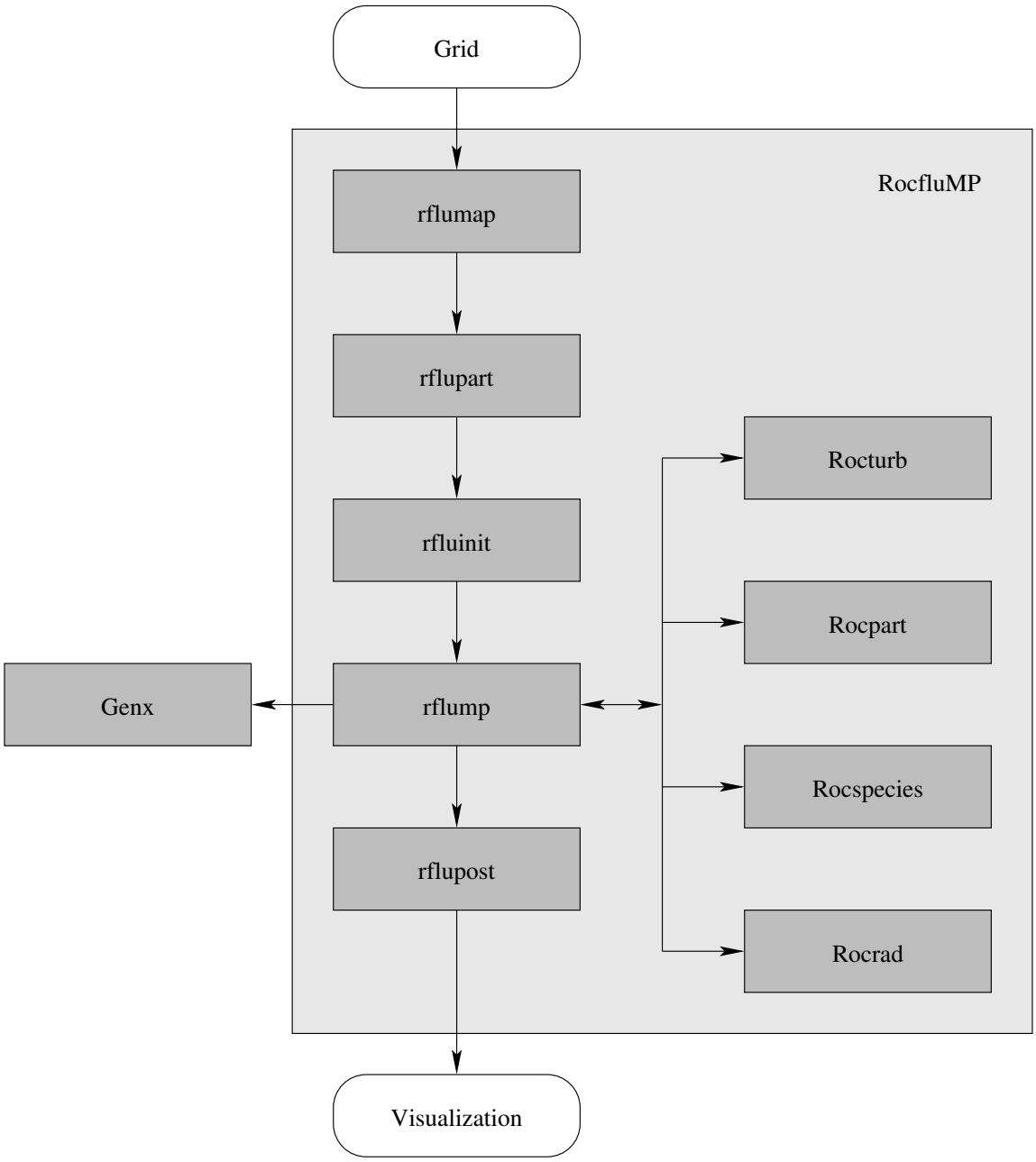

Figure 1.1: Overview of RocfluMP and related codes.

- rflupart is the partitioning module of RocfluMP. It converts grid files from outside formats into binary or ASCII files in RocfluMP format and partitions the grids into regions. At present, the following formats are supported:
  - HYBRID format (CENTAUR grid generator, CentaurSoft, Austin, TX)
  - VGRIDns format (VGRIDns grid generator, Shahyar Pirzadeh, NASA Langley)
  - MESH3D format (MESH3D grid generator, Tim Baker, Princeton University)
  - TETMESH format (TETMESH grid generator, SIMULOG, France)
  - Cobalt format (Cobalt flow solver, Cobalt Solutions LLC, Springfield, OH)
  - GAMBIT format (GAMBIT grid generator, Fluent, Lebanon, NH)

1.3 Overview of Developer's Guide

The remainder of the developer's guide consists of the following chapters:

1.4 Related Documents

The information contained in this document is supplemented by the following documents:

- "RocfluMP User's Guide" by A. Haselbacher.
- "Multiphysics Framework for RocfluidMP" by J. Ferry, F. Najjar, and S. Balachandar.
- "Charm++ FEM FrameworkManual" available at: `http://charm.cs.uiuc.edu`.
- "Charm++ Installation and Usage", available at: `http://charm.cs.uiuc.edu`.
- "Proposed Design of Interface in GEN2.5", by P.H. Geubelle, X. Jiao, and A. Haselbacher

Chapter 2

Nomenclature, Notation, and Conventions

2.1 Nomenclature

The following conventions are used in this document:

1. A *grid level*, or simply *level*, represents the entire solution domain of RocfluMP. A grid level can consist of one or more *solution regions*.

2. A *solution region*, or simply *region*, is obtained from a grid level by partitioning it for parallel processing. For sequential processing, the domain encompasses the entire grid level.

3. A *grid* is defined to be an arbitrary collection of *grid cells*, or simply *cells*.

4. A *grid cell* is defined to be a convex polyhedron of maximum degree five. Cells are categorized into *cell types*. RocfluMP only allows hexahedral, prismatic, pyramidal, and tetrahedral cell types. Each cell is composed of *faces*.

5. A *face* is defined to be a polygon of maximum degree four. RocfluMP only allows triangular or quadrilateral faces. Each face is composed of *edges*.

6. An *edge* is defined to be a straight line linking two *vertices*.

7. A *vertex* is defined to be a point in space. A vertex belongs to at least one cell. A vertex is not necessarily equivalent to a node.

2.2 Notation

E	total energy per unit mass
E^P	energy source from particles

E^S	energy source from smoke
E^{SGS}	subgrid-scale energy flux
$\vec{f}_e$	vector of external volume forces
f^P	momentum source from particles
f^S	momentum source from smoke
$\vec{F}_c$	vector of convective fluxes
$\vec{F}_v$	vector of viscous fluxes
F^R	energy flux due to radiation
H	total (stagnation) enthalpy
k	thermal conductivity coefficient
m^P	mass source from particles
n_x, n_y, n_z	components of unit normal vector in $x-, y-, z-$direction
p	static pressure
$\dot{q}_h$	heat source
$\vec{Q}$	source term
dS	surface element
t	time
u, v, w	Cartesian velocity components
$\vec{v}$	velocity vector
V	contravariant velocity
$\vec{W}$	vector of conservative variables
x, y, z	Cartesian coordinates
μ	dynamic viscosity coefficient
ρ	density
τ	viscous stress
Ω	control volume
$\partial\Omega$	boundary of a control volume

2.3 Conventions

1. SI (Système International) units are used in RocfluMP and all documents relating to RocfluMP.

2. All coordinate systems are right-handed.

3. Normal vectors point out of the solution domain.

Chapter 3

Governing Equations

3.1 The Navier-Stokes Equations

RocfluMP solves the three-dimensional time-dependent compressible Navier-Stokes equations in integral form on moving and/or deforming grids,

$$\frac{\partial}{\partial t}\int_{\Omega} \vec{W}\, d\Omega + \oint_{\partial\Omega} \vec{F}_c\, dS = \oint_{\partial\Omega} \vec{F}_v\, dS \int_{\Omega} \vec{Q}\, d\Omega. \tag{3.1}$$

The vector of the conservative variables $\vec{W}$ is

$$\vec{W} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix}. \tag{3.2}$$

The vector of convective fluxes reads

$$\vec{F}_c = \begin{bmatrix} \rho V \\ \rho u V + n_x p \\ \rho v V + n_y p \\ \rho w V + n_z p \\ \rho H V + V_g p + F^R \end{bmatrix} \tag{3.3}$$

with the face-normal velocity given by

$$V = n_x u + n_y v + n_z w - V_g, \tag{3.4}$$

where V_g is the grid speed, i.e., the grid velocity normal to the control-volume face. The vector of the viscous fluxes can be written as

$$\vec{F}_v = \begin{bmatrix} 0 \\ n_x\tau_{xx} + n_y\tau_{xy} + n_z\tau_{xz} \\ n_x\tau_{yx} + n_y\tau_{yy} + n_z\tau_{yz} \\ n_x\tau_{zx} + n_y\tau_{zy} + n_z\tau_{zz} \\ n_x\Theta_x + n_y\Theta_y + n_z\Theta_z \end{bmatrix}, \tag{3.5}$$

where

$$\begin{aligned}\Theta_x &= u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + k\frac{\partial T}{\partial x} + E_x^{SGS}\\ \Theta_y &= u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + k\frac{\partial T}{\partial y} + E_y^{SGS}\\ \Theta_z &= u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + k\frac{\partial T}{\partial z} + E_z^{SGS}\end{aligned} \tag{3.6}$$

are the terms describing the work of the viscous stresses and the heat conduction in the fluid. In the case of LES, the viscous stresses read

$$\tau_{ij} = 2\mu\, S_{i,j} - \left(\frac{2\mu}{3}\right)\frac{\partial v_k}{\partial x_k}\delta_{ij} + \tau_{ij}^{SGS} \tag{3.7}$$

with τ_{ij}^{SGS} being the subgrid stresses. In the case of RANS equations, the subgrid terms E^{SGS} in Eq. (3.6) and τ_{ij}^{SGS} in Eq. (3.7) are omitted. Instead, the dynamic viscosity and the thermal conductivity are split into a laminar and a turbulent part, i.e.,

$$\mu = \mu_L + \mu_T \tag{3.8}$$

and

$$k = k_L + k_T = c_p\left(\frac{\mu_L}{Pr_L} + \frac{\mu_T}{Pr_T}\right), \tag{3.9}$$

where c_p stands for the specific heat coefficient at constant pressure, and Pr is the Prandtl number.

The source term is given by

$$\vec{Q} = \begin{bmatrix} m^P \\ \rho f_{e,x} + f_x^P + f_x^S \\ \rho f_{e,y} + f_y^P + f_y^S \\ \rho f_{e,z} + f_z^P + f_z^S \\ \rho\vec{f_e}\cdot\vec{v} + \dot{q}_h + E^P + E^S \end{bmatrix}, \tag{3.10}$$

where m^P, f^P, f^S, E^P, and E^S represent the source terms introduced by the particle and smoke modeling. The vector of external volume forces reads

$$\vec{f_e} = \vec{g} - \vec{a} \tag{3.11}$$

with $\vec{g}$ being the gravitational acceleration and $\vec{a}$ the acceleration of the rocket.

3.2 The Geometric Conservation Law

For uniform flow, Eq. (3.1) reduces to the Geometric Conservation Law (GCL),

$$\frac{\partial}{\partial t}\int_\Omega d\Omega - \oint_{\partial\Omega} V_t\, dS = 0\,. \tag{3.12}$$

The GCL ensures that the motion of the grid does not alter a uniform flow. It must be satisfied in a discrete sense, independent of the deformation of the grid and the numerical solution method. The discretization of the GCL in RocfluMP is described in Sec. 4.5.2.

3.3 Gas Models

3.3.1 Calorically Perfect Gas

3.3.2 Thermally Perfect Gas

3.4 Thermodynamic Properties

3.5 Transport Properties

3.5.1 Viscosity

3.5.2 Conductivity

3.6 Boundary Conditions

Chapter 4

Algorithms and Methods

4.1 Geometry Definition

RocfluMP uses a method originally due to Bruner [1] and improved by Wang [2] to compute geometrical properties of faces and volumes. The method is particularly convenient for finite-volume schemes because volume properties are expressed in terms of face properties. This means that the face and volume properties can be computed in a single loop over faces.

4.1.1 Computation of Face Properties

The face properties of interest are the normal vector, the area, and its centroid.

For a triangular face, the scaled normal vector is given by

$$\mathbf{n} = \frac{1}{2}\left(\mathbf{r}_{12} \times \mathbf{r}_{23}\right). \tag{4.1}$$

where $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$. For a quadrilateral face, the normal vector is given by the average of the average normal vectors obtained by subdividing the face into two triangles.

The area of triangular and quadrilateral faces follows from Eq. 4.1 as

$$S = \|\mathbf{n}\|_2. \tag{4.2}$$

The centroid of a triangular face is computed from

$$\mathbf{r}^c = \frac{1}{3}\left(\mathbf{r}_1 + \mathbf{r}_2 + \mathbf{r}_3\right). \tag{4.3}$$

For a quadrilateral face, the centroid is given by the average of the average centroids obtained by subdividing the face into two triangles.

4.1.2 Computation of Volume Properties

The volume properties of interest are the volume and its centroid.

For a polyhedron composed of N triangular faces, the volume may be computed from

$$V = \frac{1}{3} \sum_{i=1}^{N} \mathbf{r}_i^c \cdot \mathbf{n}_i \, S_i. \tag{4.4}$$

The centroid of a polyhedron composed of N triangular faces may be computed from

$$\mathbf{r}_c = \frac{1}{4V} \sum_{i=1}^{N} (\mathbf{r}_i^c \cdot \mathbf{n}_i) \; \mathbf{r}_i^c \, S_i \tag{4.5}$$

For polyhedra with quadrilateral faces, Eqs. (4.4) and (4.5) can be applied given that they only involve face properties which have already been computed.

It is interesting to note that Eq. (4.5) expresses the volume centroid as a weighted sum of face centroids, and that the weights are not guaranteed to be positive. Positive weights can be guaranteed by first computing an approximate cell centroid $\tilde{\mathbf{r}}_c$ as the average of the vertex position vectors, and then replacing $\mathbf{r}_i^c \cdot \mathbf{n}_i$ by $(\mathbf{r}_i^c - \widetilde{\mathbf{r}}_c) \cdot \mathbf{n}_i$ in Eq. (4.5).

4.2 Spatial Discretization

4.2.1 Stencil Construction

Explicit stencils only need to be constructed for the interpolation and gradient operators. The minimum extent or size of these stencils is determined by their order of accuracy.

For an interpolation operator of order p on an arbitrary grid, the minimum extent is given by

$$\overline{N}_{\min} = \frac{(p+1)(p+2)(p+3)}{6} + 1. \tag{4.6}$$

In the present work, filter operators are regarded as low-order interpolation operators. For a gradient operator of order q on an arbitrary grid, the minimum extent is given by

$$N_{\min} = \frac{(q+1)(q+2)(q+3)}{6}. \tag{4.7}$$

For both interpolation and gradient operators, it may be advantageous to increase the stencil extent to include more cell centroids than necessary. For filter operators, this can be used to minimize the imaginary part of the transfer function. The larger-than-necessary support necessitates the use of least-squares techniques to determine the interpolated value or gradients.

Gradient operators are required at cell and face centroids. Interpolation operators are required at cell and face centroids and at vertices. Hence cell-to-cell, face-to-cell, and vertex-to-cell stencils must be constructed, as depicted schematically in Fig. 4.1.

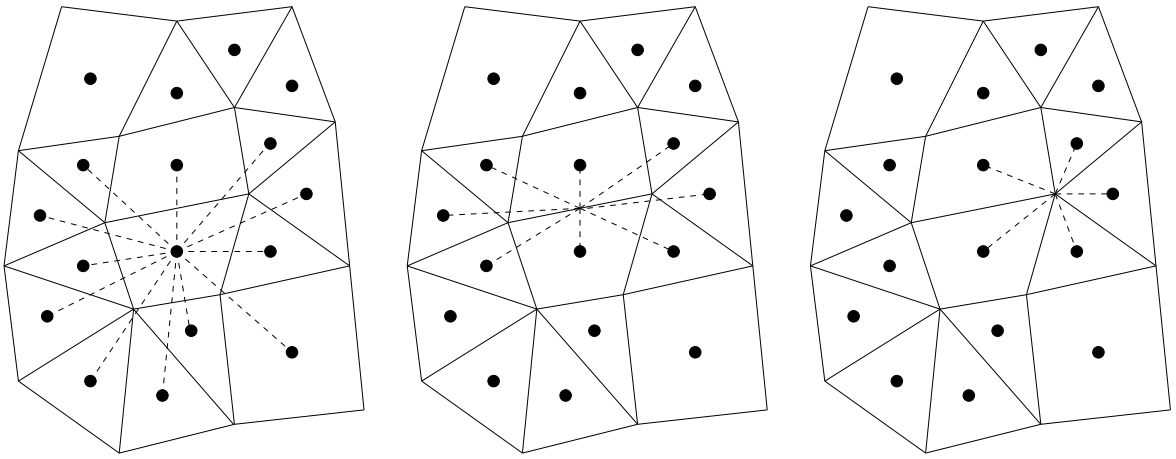

Figure 4.1: Schematic illustration stencils in two dimension. **(a)** Cell-to-cell stencil, **(b)** face-to-cell stencil, and **(c)** vertex-to-cell stencil.

At present, these stencils are constructed using an Octree-based approach. The Octree is initialized using cell-centroid coordinates, and queried with the locations at which the interpolation or gradient operators are to be constructed.

4.2.2 Interpolation Operators

The interpolation operators are constructed using a least-squares approach based on a modified Taylor series. Assuming linear interpolation of a scalar variable ϕ, this gives

$$\phi_i = \overline{\phi}_0 + (\nabla\phi)_0 \cdot \Delta\mathbf{r}_{0i}, \tag{4.8}$$

where $\overline{\phi}_0$ is the interpolated value at location 0, which may be a cell or face centroid or a vertex. Assuming that Eq. (4.8) is applied to the d_0 points in the stencil, an overdetermined system of linear equations is obtained,

$$\begin{bmatrix} \Delta x_{01} & \Delta y_{01} & 1 \\ \Delta x_{02} & \Delta y_{02} & 1 \\ \vdots & \vdots & \vdots \\ \Delta x_{0d_0} & \Delta y_{0d_0} & 1 \end{bmatrix} \begin{Bmatrix} \partial_x\phi \\ \partial_y\phi \\ \overline{\phi} \end{Bmatrix}_0 = \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{d_0} \end{Bmatrix}, \tag{4.9}$$

or

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{4.10}$$

The system can be inverted using the Singular Value Decomposition (SVD), which gives a interpolation formula of the form,

$$\overline{\phi}_0 = \sum_{i=1}^{d_0} \overline{\omega}_{0i}\phi_i, \tag{4.11}$$

where the stencil weights are given by

$$\overline{\omega}_{0i} = \mathbf{A}^{-1}_{3,i} \quad \text{for} \quad 1 \leq i \leq d_0. \tag{4.12}$$

4.2.3 Gradient Operators

The gradient operators are also constructed using a least-squares approach. Assuming linear gradient reconstruction of a scalar variable ϕ, this gives

$$\phi_i = \phi_0 + (\nabla\phi)_0 \cdot \Delta\mathbf{r}_{0i}, \tag{4.13}$$

where ϕ_0 is now the value at location 0, which may be a cell or face centroid or a vertex. Assuming that Eq. (4.13) is applied to the d_0 points in the stencil, an overdetermined system of linear equations is obtained,

$$\begin{bmatrix} \Delta x_{01} & \Delta y_{01} \\ \Delta x_{02} & \Delta y_{02} \\ \vdots & \vdots \\ \Delta x_{0d_0} & \Delta y_{0d_0} \end{bmatrix} \begin{Bmatrix} \partial_x\phi \\ \partial_y\phi \end{Bmatrix}_0 = \begin{Bmatrix} \Delta\phi_{01} \\ \Delta\phi_{02} \\ \vdots \\ \Delta\phi_{0d_0} \end{Bmatrix}, \tag{4.14}$$

or

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{4.15}$$

The system can be inverted using the Singular Value Decomposition (SVD), which gives a formula of the form,

$$(\nabla\phi)_0 = \sum_{i=1}^{d_0} \boldsymbol{\omega}_{0i}\Delta\phi_{0i}, \tag{4.16}$$

where the vector of stencil weights is given by

$$\boldsymbol{\omega}_{0i} = \mathbf{A}^{-1}_{1:2,i} \quad \text{for} \quad 1 \leq i \leq d_0. \tag{4.17}$$

4.2.4 Inviscid Fluxes

4.2.4.1 Limiter Functions

4.2.4.2 Numerical Flux Functions

4.2.4.3 Entropy Fixes

4.2.5 Viscous Fluxes

4.2.6 Optimal LES Discretization

4.2.6.1 Computation of Integrals

4.2.6.2 Computation of Stencil Weights

4.2.7 Source Terms

4.3 Boundary Conditions

4.3.1 Fluid-Solid Boundary Conditions

4.3.1.1 Slip Wall Boundaries

4.3.1.2 No-Slip Wall Boundaries

4.3.1.3 Injection Wall Boundaries

4.3.2 Fluid-Fluid Boundary Conditions

4.3.2.1 Inflow Boundaries

4.3.2.2 Outflow Boundaries

4.3.2.3 Periodic Boundaries

4.3.2.4 Symmetry Boundaries

4.4 Temporal Discretization

4.4.1 Runge-Kutta Methods

4.4.2 Computation of Time Step

4.5 Grid Motion

4.5.1 Grid Smoothing

4.5.2 Discrete Geometric Conservation Law

4.5.3 Implementation Details

4.6 Pressure, Skin-Friction, and Heat-Transfer Coefficient Computation

$\mathbf{n}_i$.

These pressure coefficient is defined as

$$C_{p,i} = \frac{p_i - p_{\text{ref}}}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2}. \tag{4.18}$$

The skin-friction coefficients are defined as

$$C_{fx,i} = \frac{(\boldsymbol{\tau}_i \cdot \mathbf{n}_i)_x}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2}, \tag{4.19a}$$

$$C_{fy,i} = \frac{(\boldsymbol{\tau}_i \cdot \mathbf{n}_i)_y}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2}, \tag{4.19b}$$

$$C_{fz,i} = \frac{(\boldsymbol{\tau}_i \cdot \mathbf{n}_i)_z}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2}. \tag{4.19c}$$

where $\boldsymbol{\tau}_i \cdot \mathbf{n}_i$ is the viscous stress acting on the face.

The heat-transfer coefficient is defined as

$$C_{h,i} = \frac{\mathbf{q}_i \cdot \mathbf{n}_i}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^3}. \tag{4.20}$$

where $\mathbf{q}_i$ is the heat flux for the face.

4.7 Force and Moment Computation

RocfluMP computes forces and moments exerted by the fluid on the patches. The force and moment on a patch are computed from the sum of the forces and moments on the faces of that patch.

The force on a face i with unit normal vector $\mathbf{n}_i$ and area S_i is

$$\mathbf{F}_i = \left[(p_i - p_{\text{ref}})\,\mathbf{n}_i - \boldsymbol{\tau}_i \cdot \mathbf{n}_i\right] S_i, \tag{4.21}$$

where p_i and $\boldsymbol{\tau}_i \cdot \mathbf{n}_i$ are the pressure and viscous stress acting on the face, respectively. The force components are given by

$$F_{x,i} = \Big[(p_i - p_{\text{ref}})\,n_{x,i} - (\boldsymbol{\tau}_i \cdot \mathbf{n}_i)_x\Big] S_i, \tag{4.22a}$$

$$F_{y,i} = \Big[(p_i - p_{\text{ref}})\,n_{y,i} - (\boldsymbol{\tau}_i \cdot \mathbf{n}_i)_y\Big] S_i, \tag{4.22b}$$

$$F_{z,i} = \Big[(p_i - p_{\text{ref}})\,n_{z,i} - (\boldsymbol{\tau}_i \cdot \mathbf{n}_i)_z\Big] S_i. \tag{4.22c}$$

The moment about a reference location $\mathbf{r}_{\text{ref}}$ created by the force on a face is

$$\mathbf{M}_i = (\mathbf{r}_i - \mathbf{r}_{\text{ref}}) \times \mathbf{F}_i, \tag{4.23}$$

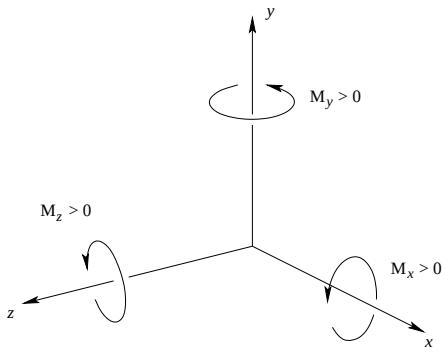

Figure 4.2: Definition for positive moments.

where $\mathbf{r}_i$ is the position vector of the face centroid. The convention for positive moments is shown in Fig. 4.2: Moments around a given coordinate axis are defined to be positive if they lead to a counter-clockwise rotation when looking in the negative direction of that coordinate axis. The moments components are given by

$$M_{x,i} = F_{z,i}(y_i - y_{\text{ref}}) - F_{y,i}(z_i - z_{\text{ref}}), \tag{4.24a}$$
$$M_{y,i} = F_{x,i}(z_i - z_{\text{ref}}) - F_{z,i}(x_i - x_{\text{ref}}), \tag{4.24b}$$
$$M_{z,i} = F_{y,i}(x_i - x_{\text{ref}}) - F_{x,i}(y_i - y_{\text{ref}}). \tag{4.24c}$$

Non-dimensional force coefficients are defined by

$$C_{F_{x,i}} = \frac{F_{x,i}}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2 S_{\text{ref}}} = \Big(C_{p,i} n_{x,i} - C_{fx,i}\Big)\frac{S_i}{S_{\text{ref}}}, \tag{4.25a}$$
$$C_{F_{y,i}} = \frac{F_{y,i}}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2 S_{\text{ref}}} = \Big(C_{p,i} n_{y,i} - C_{fy,i}\Big)\frac{S_i}{S_{\text{ref}}}, \tag{4.25b}$$
$$C_{F_{z,i}} = \frac{F_{z,i}}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2 S_{\text{ref}}} = \Big(C_{p,i} n_{z,i} - C_{fz,i}\Big)\frac{S_i}{S_{\text{ref}}}, \tag{4.25c}$$

where the pressure coefficient $C_{p,i}$ is defined by Eq. (4.18) and the skin-friction coefficients $C_{fx,i}$, $C_{fy,i}$, and $C_{fz,i}$ are defined by Eqs. (4.19).

Non-dimensional moment coefficients are defined by

$$C_{M_{x,i}} = \frac{M_{x,i}}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2 S_{\text{ref}} L_{\text{ref}}} = C_{F_{z,i}}\frac{y_i - y_{\text{ref}}}{L_{\text{ref}}} - C_{F_{y,i}}\frac{z_i - z_{\text{ref}}}{L_{\text{ref}}}, \tag{4.26a}$$
$$C_{M_{y,i}} = \frac{M_{y,i}}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2 S_{\text{ref}} L_{\text{ref}}} = C_{F_{x,i}}\frac{z_i - z_{\text{ref}}}{L_{\text{ref}}} - C_{F_{z,i}}\frac{x_i - x_{\text{ref}}}{L_{\text{ref}}}, \tag{4.26b}$$
$$C_{M_{z,i}} = \frac{M_{z,i}}{\frac{1}{2}\rho_{\text{ref}} V_{\text{ref}}^2 S_{\text{ref}} L_{\text{ref}}} = C_{F_{y,i}}\frac{x_i - x_{\text{ref}}}{L_{\text{ref}}} - C_{F_{x,i}}\frac{y_i - y_{\text{ref}}}{L_{\text{ref}}}. \tag{4.26c}$$

With the definitions of the force coefficients, the moment coefficients may be written as

$$C_{M_{x,i}} = \left[\left(C_{p,i}n_{z,i} - C_{fz,i}\right)\frac{y_i - y_{\text{ref}}}{L_{\text{ref}}} - \left(C_{p,i}n_{y,i} - C_{fy,i}\right)\frac{z_i - z_{\text{ref}}}{L_{\text{ref}}}\right]\frac{S_i}{S_{\text{ref}}}, \tag{4.27a}$$

$$C_{M_{y,i}} = \left[\left(C_{p,i}n_{x,i} - C_{fx,i}\right)\frac{z_i - z_{\text{ref}}}{L_{\text{ref}}} - \left(C_{p,i}n_{z,i} - C_{fz,i}\right)\frac{x_i - x_{\text{ref}}}{L_{\text{ref}}}\right]\frac{S_i}{S_{\text{ref}}}, \tag{4.27b}$$

$$C_{M_{z,i}} = \left[\left(C_{p,i}n_{y,i} - C_{fy,i}\right)\frac{x_i - x_{\text{ref}}}{L_{\text{ref}}} - \left(C_{p,i}n_{x,i} - C_{fx,i}\right)\frac{y_i - y_{\text{ref}}}{L_{\text{ref}}}\right]\frac{S_i}{S_{\text{ref}}}. \tag{4.27c}$$

The force and moment coefficients for an entire patch are simply given by the summation of the force and moment coefficients for the faces on that patch,

$$C_{F_x} = \frac{1}{S_{\text{ref}}}\sum_i \left(C_{p,i}n_{x,i} - C_{fx,i}\right)S_i, \tag{4.28a}$$

$$C_{F_y} = \frac{1}{S_{\text{ref}}}\sum_i \left(C_{p,i}n_{y,i} - C_{fy,i}\right)S_i, \tag{4.28b}$$

$$C_{F_z} = \frac{1}{S_{\text{ref}}}\sum_i \left(C_{p,i}n_{z,i} - C_{fz,i}\right)S_i, \tag{4.28c}$$

and

$$C_{M_x} = \frac{1}{S_{\text{ref}}L_{\text{ref}}}\sum_i \left[\left(C_{p,i}n_{z,i} - C_{fz,i}\right)(y_i - y_{\text{ref}}) - \left(C_{p,i}n_{y,i} - C_{fy,i}\right)(z_i - z_{\text{ref}})\right]S_i, \tag{4.29a}$$

$$C_{M_y} = \frac{1}{S_{\text{ref}}L_{\text{ref}}}\sum_i \left[\left(C_{p,i}n_{x,i} - C_{fx,i}\right)(z_i - z_{\text{ref}}) - \left(C_{p,i}n_{z,i} - C_{fz,i}\right)(x_i - x_{\text{ref}})\right]S_i, \tag{4.29b}$$

$$C_{M_z} = \frac{1}{S_{\text{ref}}L_{\text{ref}}}\sum_i \left[\left(C_{p,i}n_{y,i} - C_{fy,i}\right)(x_i - x_{\text{ref}}) - \left(C_{p,i}n_{x,i} - C_{fx,i}\right)(y_i - y_{\text{ref}})\right]S_i. \tag{4.29c}$$

Chapter 5

Code Design and Organization

RocfluMP has been designed to share code with its block-structured sister code RocfloMP. The code sharing is reflected in both the directory structure in which the source files for RocfluMP are stored as well as the flow of control within RocfluMP. The directory structure and the control flow within RocfluMP are described in this chapter.

5.1 Directory Structure

Code sharing is reflected in the directory structure because shared routines and modules are placed in dedicated directories. A schematic overview of the most important directories and how the executables are built for the standalone and coupled cases is shown in Fig. 5.1.

A detailed description of the directories relevant to RocfluMP is as follows:

`genx` Contains rflump (and rflomp) routines which are used only within GENx. On compilation, the routines are linked with the library `libRFLU.a` to produce `libflu.a`, which in turn is linked with other libraries into the executable `genx.x`.

`libfloflu` Contains RocfluMP (and RocfloMP) routines. On compilation, the routines are compiled with the modules from `modfloflu` and `modflu` into the library `libFLOFLU.a`. The library is used by rflump (and rflomp), rfluprep, and rflupost.

`libflu` Contains only RocfluMP routines. On compilation, the routines are compiled with the modules from `modfloflu` and `modflu` into the library `libFLU.a`. The library is used by rflump, rfluprep, and rflupost.

`modfloflu` Contains RocfluMP (and RocfloMP) modules. The modules are used by rflump (and rflomp), rfluprep, and rflupost.

`modflu` Contains only RocfluMP modules. The modules are used by rflump, rfluprep, and rflupost.

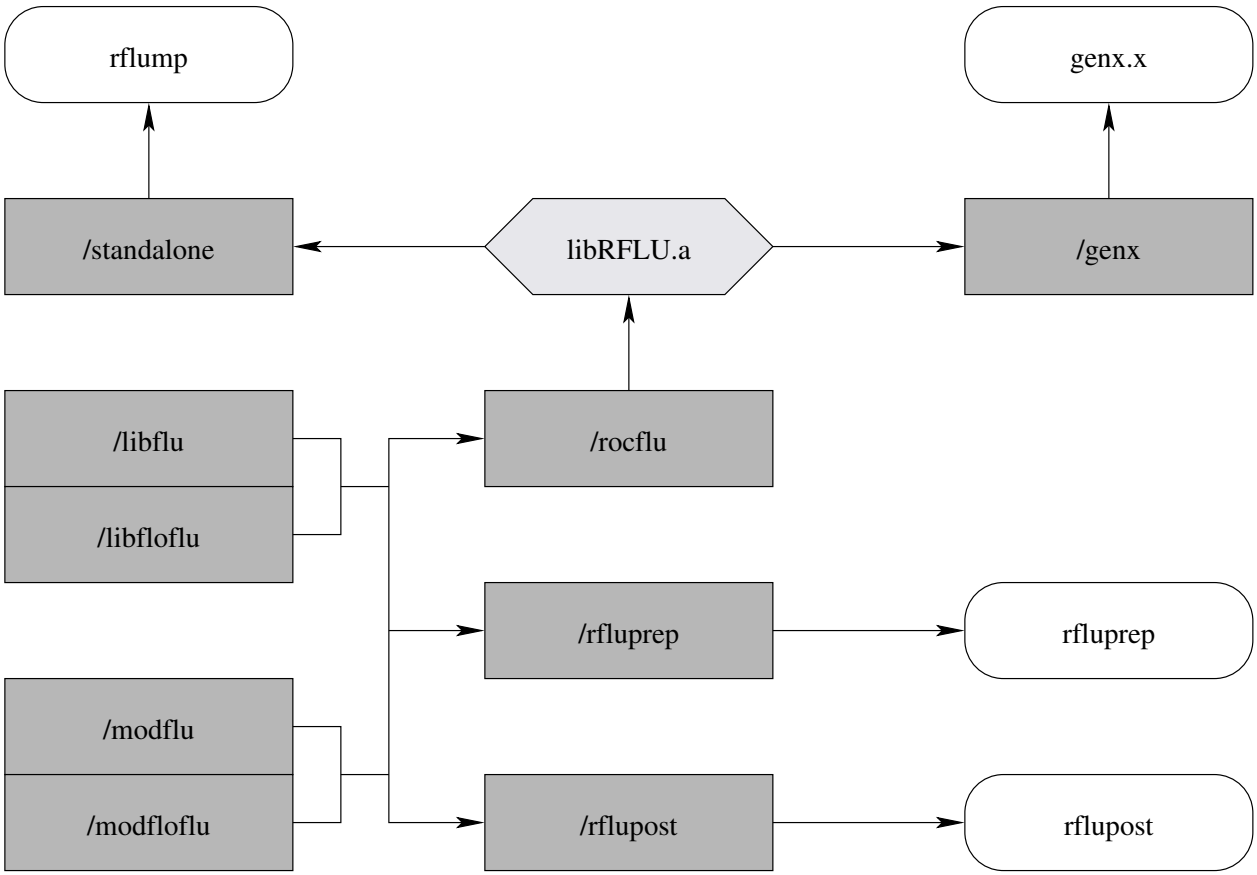

Figure 5.1: Schematic overview of directory structure relevant to RocfluMP and building of executables.

`rocflu` Contains RocfluMP routines for the actual flow solver. The routines are only used by rflump. On compilation, the routines are compiled with the modules from `modfloflu` and `modflu` and linked with `libFLU.a` and `libFLOFLU.a` to produce the executable `libRFLU.a`.

`standalone` Contains rflump (and rflomp) routines which drive rflump. The `main.F90` routine is located in this directory. On compiling rflump into a standalone code, these routines are linked with `libRFLU.a` to produce the executable `rflump`.

`utilities/rocflu/prep` Contains only rfluprep routines. On compilation, the routines are compiled with the modules from `modfloflu` and `modflu` and are linked with `libFLU.a` and `libFLOFLU.a` to produce the executable `rfluprep`.

`utilities/rocflu/post` Contains only rflupost routines. Contains only rfluprep routines. On compilation, the routines are compiled with the modules from `modfloflu` and `modflu` and are linked with `libFLU.a` and `libFLOFLU.a` to produce the executable `rflupost`.

5.2 Control Flow

The following sections outline the control flow of rflump. Attention is restricted to the top-level routines; the control flow of lower-level routines is best studied directly in source-code form. Because the invokation of the top-level routines is slightly different depending on whether rflump is run as a standalone code or coupled with GENx, these two cases are discussed separately.

5.2.1 Standalone Code

On running rflump, the `main.F90` routine calls the routines `RFLU_InitFlowSolver.F90`, `RFLU_FlowSolver.F90`, and `RFLU_EndFlowSolver.F90`. The names of these routines are self-explanatory.

In the following, attention is focused on the first two of the above routines; the routine `RFLU_EndFlowSolver.F90` does not merit discussion and is best studied directly in source-code form.

The control flow of the routine `RFLU_InitFlowSolver.F90` is depicted schematically in Fig. 5.2. It is worth noting that the grid and solution files are read in together. This is because in coupled simulations, Rocman requires that the solution variables are known when the data fields are registered (this step is not shown in Fig. 5.2). This means that the memory usage is somewhat higher than would be necessary for non-coupled simulations, because the solution is stored while temporary memory is allocated for the construction of data structures.

The control flow of the routine `RFLU_FlowSolver.F90` is depicted schematically in Fig. 5.3. The actual stepping, either in terms of time for time-accurate computations or in terms

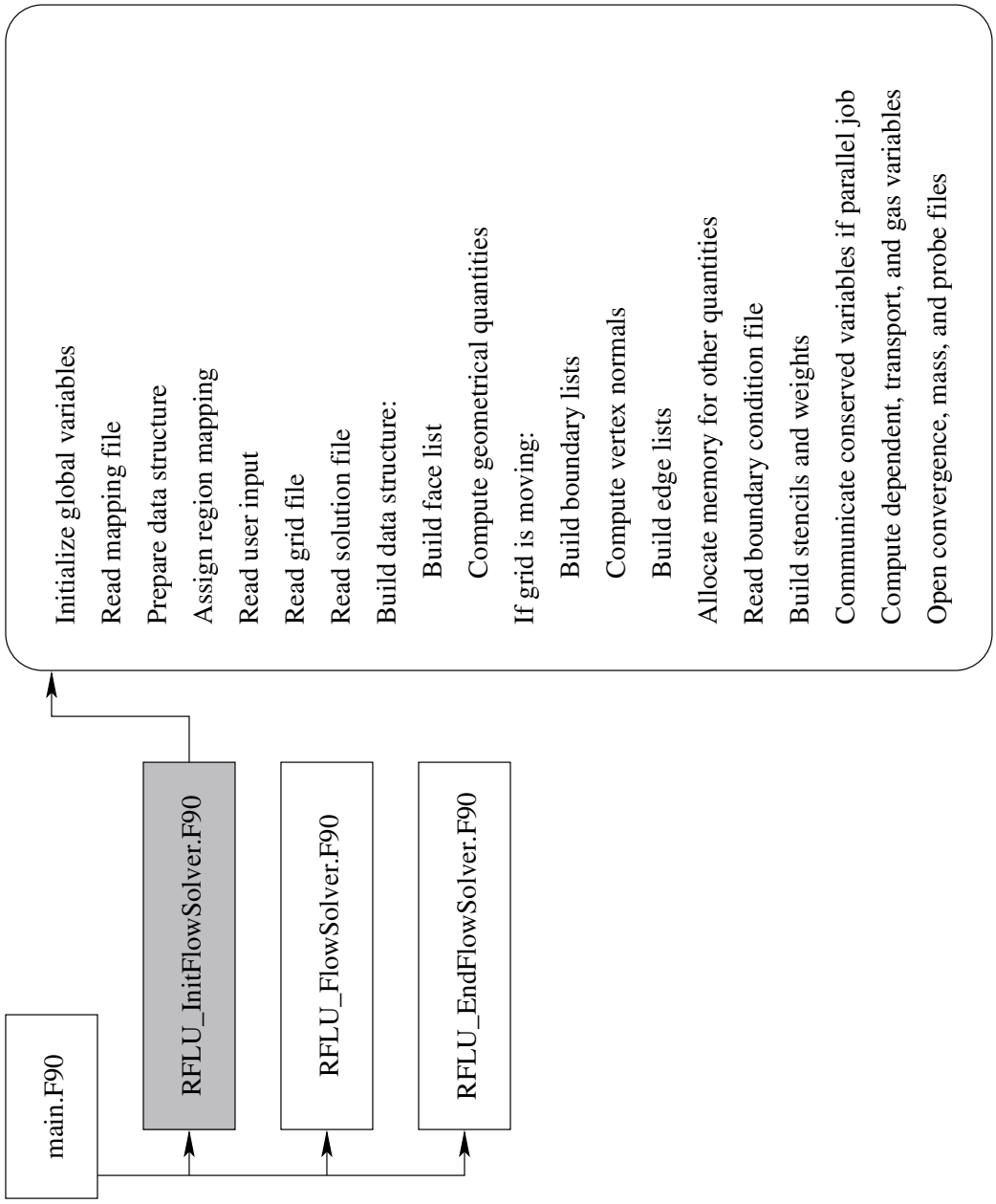

Figure 5.2: Overview of control flow in top-level routines of standalone code with focus on pre-processing.

of iterations for steady-state computations, is carried out in `RFLU_TimeStepping.F90`. The solution is evolved by one time step in the routines `rungeKutta.F90` for unsteady flows or `explicitMultiStage.F90` for steady flows.

5.2.2 Code Coupled With GENx

5.3 Error Handling

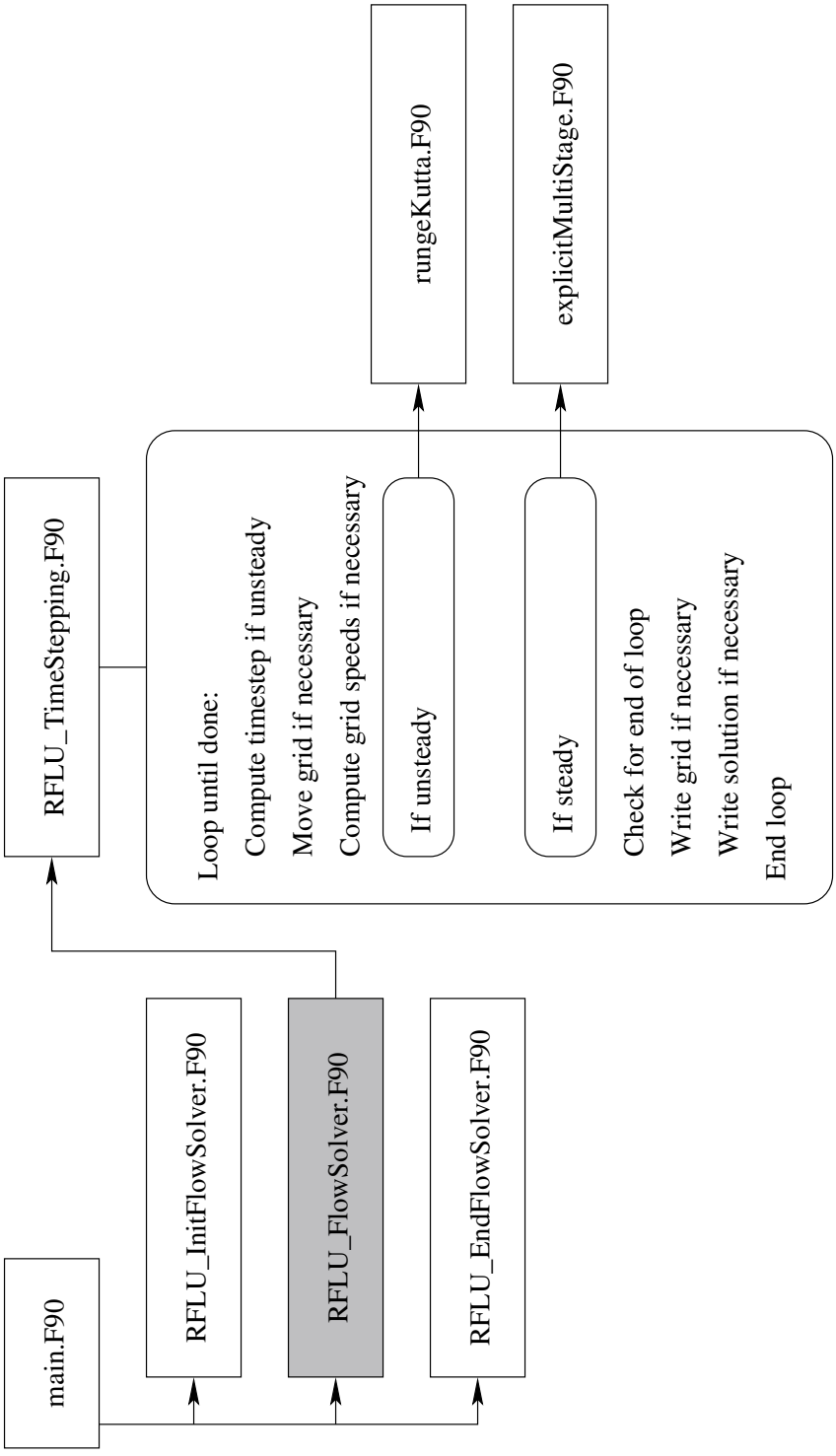

Figure 5.3: Overview of control flow in top-level routines of standalone code with focus on flow-solution process.

Chapter 6

Data Structures

RocfluMP makes extensive use of Fortran 90 user-defined types for the definition of data structures.

6.1 Philosophy and Abstraction

The top layer of the data structure developed for RocfluMP is depicted schematically in Fig. 6.1.

The top layer of the data structure consists of two main layers of abstraction:

1. At the highest layer are multigrid levels constructed from the finest grid. Each layer can contain an arbitrary number of regions.

2. At the second level are solution region. A solution region is defined as the entire solution region for sequential processing applications, or a single partition for parallel processing applications. Each multigrid level can consist of an arbitrary number of domains.

Note that the multigrid levels are located atop the solution regions. This means that each multigrid level is partitioned separately for parallel processing. Because intra-layer commu-

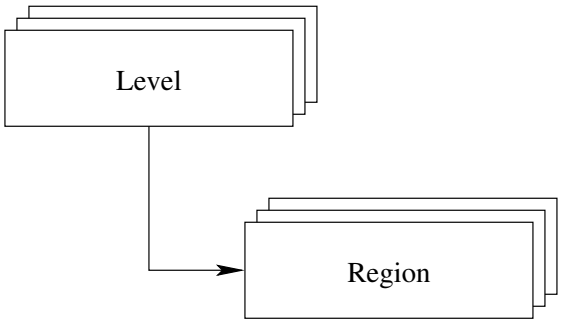

Figure 6.1: Overview of data-structure layers.

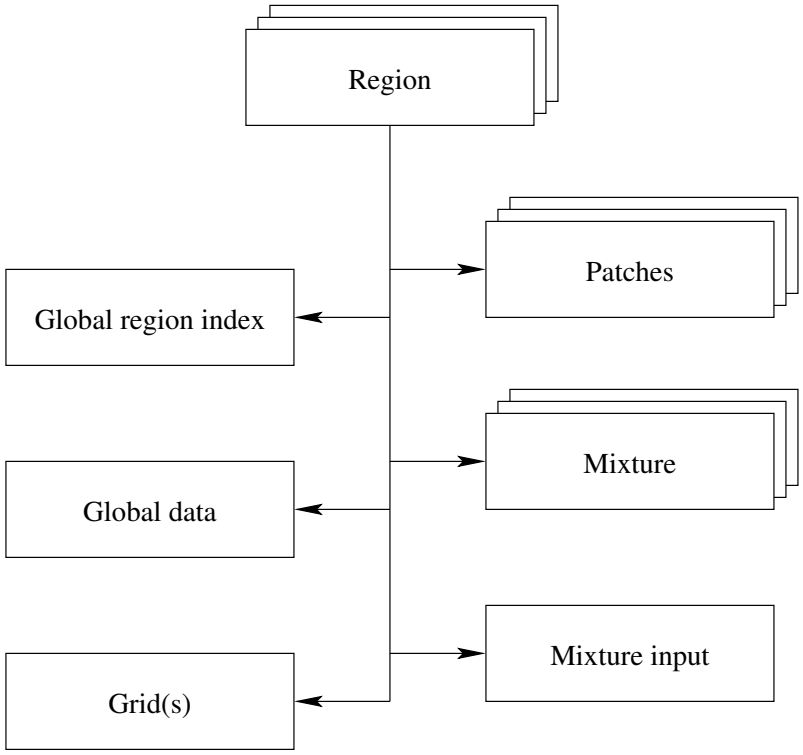

Figure 6.2: Overview of region data structure.

nication is more important than inter-level communication, the possibility of optimizing each multigrid level for load balancing separately should allow for better parallel performance.

The types `t_level` and `t_region` are defined in `ModDataStruct.F90`.

6.2 Region Data Structure

The region data structure contains all the information required to solve the governing equations in a given region. A schematic overview of the region data structure is given in Fig. 6.2.

The region data structure itself is defined to be an array. This gives additional flexibility in allowing several regions to be assigned to a single processor.

The components of the user-defined data type `t_region` are defined as follows:

`iRegionGlobal` is the global index of the local region.

`irkStep` is the index of the Runge-Kutta step.

`fieldFlagMixt` is a field flag used to communicate the conserved variables for parallel calculations using the FEM framework.

`dt` contains the timestep.

```
TYPE t_region
  INTEGER :: iRegionGlobal,irkStep
  INTEGER :: fieldFlagMixt

  REAL(RFREAL), POINTER :: dt(:)

  TYPE(t_grid) :: grid,gridOld
  TYPE(t_mixt) :: mixt
  TYPE(t_turb) :: turb
  TYPE(t_spec) :: spec
  TYPE(t_radi) :: radi
  TYPE(t_peul) :: peul

  TYPE(t_plag)  , POINTER :: plags(:)
  TYPE(t_patch) , POINTER :: patches(:)
  TYPE(t_global), POINTER :: global

  TYPE(t_mixt_input) :: mixtInput
  TYPE(t_turb_input) :: turbInput
  TYPE(t_spec_input) :: specInput
  TYPE(t_peul_input) :: peulInput
  TYPE(t_plag_input) :: plagInput
  TYPE(t_radi_input) :: radiInput
END TYPE t_region
```

Figure 6.3: Definition of region data structure.

`grid` is the user-defined data type containing all the information relating to the grid. See Sect. 6.3 for a description of `t_grid`.

`gridOld` is the user-defined data type containing all the information relating to the old grid when using grid motion. See Sect. 6.3 for a description of `t_grid`.

`mixt` is the user-define data type containing all the information relating to mixture. It is described in Sec. 6.5.2.

`patches` is the user-defined data type containing all the information relating to boundary patches. See Sect. 6.4 for a description of `t_patch`.

`global` is a pointer to global data.

`mixtInput` is a user-defined data type containing all the user-defined input for the solution of the mixture equations. It is described in Sec. 6.5.1.

6.3 Grid Data Structure

The grid data structure contains information relating to the description of the grid. An overview of the grid data structure is given in Fig. 6.4. The grid data structure is defined in `ModGrid.F90`. Some additional (RocfluMP-specific) data is defined in `RFLU_ModGrid.F90`, which is mainly used in converting from exterior grid formats to that used by RocfluMP, and in helping to construct some data structures. The two modules are discussed in detail below.

6.3.1 Module `ModGrid.F90`

In understanding the grid data structure, the following points are important:

1. RocfluMP can operate on grids consisting of arbitrary combinations of tetrahedral, hexahedral, prismatic, and pyramidal cells. As indicated in Sec. 2.1, these are referred to as instances of different *cell types*. When running RocfluMP in parallel, one also has to distinguish between actual and virtual cells, so RocfluMP introduces the concept of a *cell kind* to distinguish between these.

2. RocfluMP categorizes faces according to types and kinds also. A face can be of triangular of quadrilateral type, and can be of different kinds depending on whether the adjacent cells are actual or virtual ones, and whether the face is on a boundary.

3. Since RocfluMP is based on the cell-centered method, the computation of fluxes is most easily carried out by looping over faces of the grid. Because boundary conditions are conveniently enforced by modifying the computation of fluxes on boundary patches, the grid data structure only stores internal faces, i.e., faces which do not lie on boundary patches.

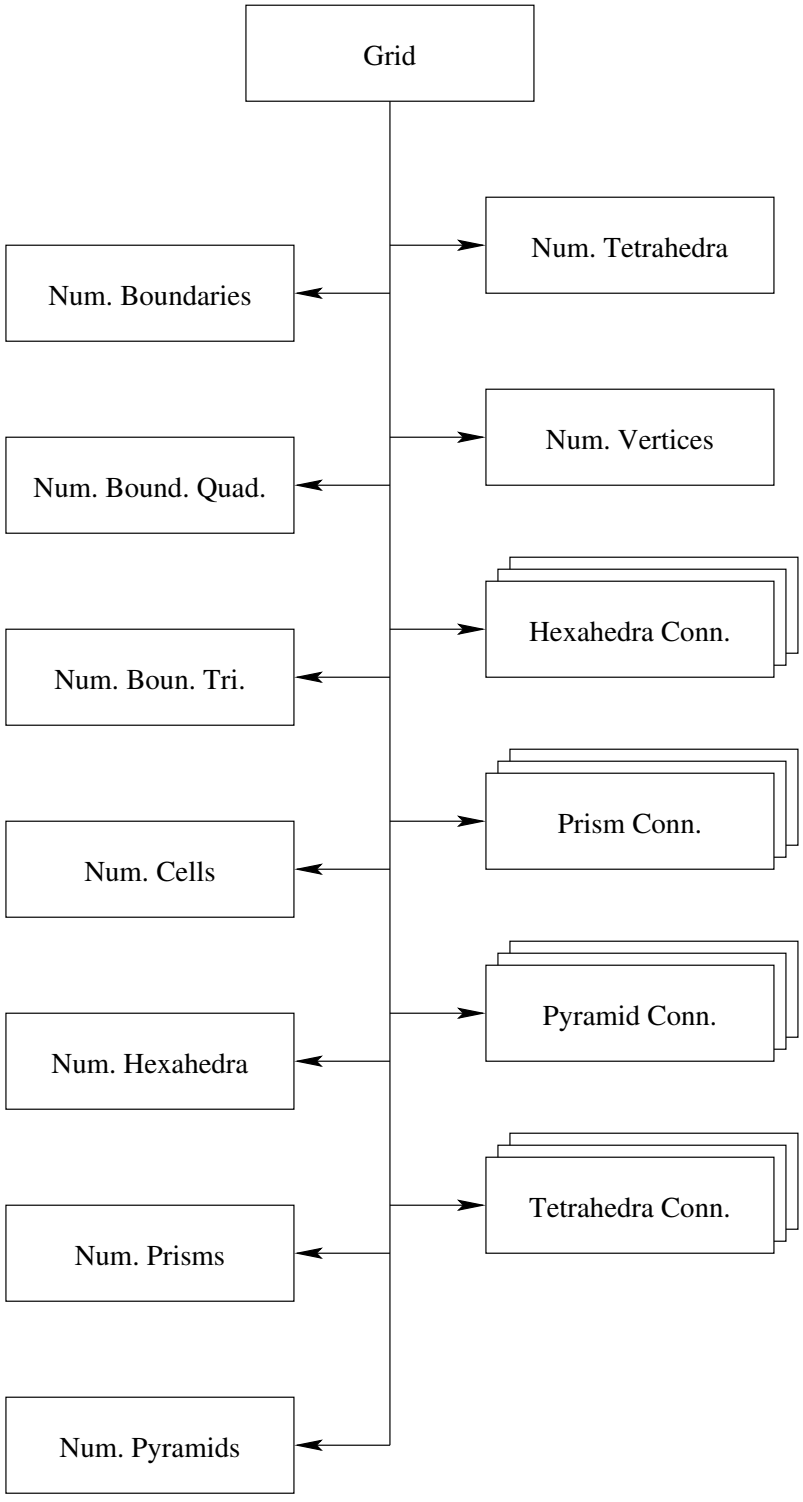

Figure 6.4: Overview of grid data structure.

```
  TYPE t_grid
! - Basic grid quantities ------------------------------------------------------

    INTEGER :: indGs,nBFaces,nBQuads,nBTris,nCells,nCellsTot,nEdges, &
               nEdgesEst,nEdgesTot,nFaces,nFacesEst,nFacesTot,nHexs, &
               nHexsTot,nPatches,nPris,nPrisTot,nPyrs,nPyrsTot, &
               nTets,nTetsTot,nVert,nVertTot
    INTEGER, DIMENSION(:), POINTER :: hexFlag,hex2CellGlob, &
                                      priFlag,pri2CellGlob,pyrFlag, &
                                      pyr2CellGlob,tetFlag,tet2CellGlob, &
                                      vertFlag,v2c
    INTEGER, DIMENSION(:,:), POINTER :: cellGlob2Loc,c2cs,e2v,e2vTemp,f2c, &
                                        f2cTemp,f2cs,f2v,f2vTemp,hex2v,pri2v, &
                                        pyr2v,tet2v,v2cInfo
    REAL(RFREAL), DIMENSION(:,:), POINTER :: fc,fn
    REAL(RFREAL), DIMENSION(:,:,:), POINTER :: cgwt,fgwt

! - Grid Motion -----------------------------------------------------------------

    INTEGER, DIMENSION(:), POINTER :: degr
    REAL(RFREAL), DIMENSION(:), POINTER :: gs,volMin
    REAL(RFREAL), DIMENSION(:,:), POINTER :: rhs

! - Geometric information -------------------------------------------------------

    REAL(RFREAL), POINTER :: xyz(:,:)
    REAL(RFREAL), POINTER :: vol(:),cofg(:,:)
  END TYPE t_grid
```

Figure 6.5: Definition of grid data structure.

The components of the user-defined type `t_grid` are defined as follows:

`indGs` is a flag used to allocate the array for the grid speeds. If grid motion is active, the grid speeds need to be computed, and hence `indGs=1`, otherwise `indGs=0`. This allows the grid speed array `gs` (see below) to be accessed even if grid motion is not active, which simplifies the code because conditional statements can be avoided. The array `gs` will typically be accessed through a statement such as `gs(indGs*ifc)`, where `ifc` is an integer variable used in a loop over interior faces.

`nBFaces` is the total number of triangular and quadrilateral faces on all boundary patches.

`nBQuads` is the total number of quadrilateral faces on all boundary patches.

`nBTris` is the total number of triangular faces on all boundary patches.

`nCells` is the number of interior cells in the grid.

`nCellsTot` is the total number of cells in the grid, i.e., interior and dummy cells.

`nEdges` is the number of interior edges in the grid.

`nEdgesEst` is the estimated total number of edges in the grid. It is used only in the construction of the edge list.

`nEdgesTot` is the total number of edges in the grid, i.e., interior and dummy edges.

`nFaces` is the number of interior triangular and quadrilateral faces in the grid.

`nFacesEst` is the estimated total number of interior triangular and quadrilateral faces in the grid. It is used only in the construction of the face list.

`nFacesTot` is the total number of triangular and quadrilateral faces in the grid, i.e., interior and dummy faces.

`nHexs` is the number of interior hexahedral cells in the grid.

`nHexsTot` is the total number of hexahedral cells in the grid, i.e., interior and dummy hexahedral cells.

`nPris` is the number of prismatic cells in the grid.

`nPrisTot` is the total number of prismatic cells in the grid, i.e., interior and dummy prismatic cells.

`nPyrs` is the number of pyramidal cells in the grid.

`nPyrsTot` is the total number of pyramidal cells in the grid, i.e., interior and dummy pyramidal cells.

`nTets` is the number of tetrahedral cells in the grid.

`nTetsTot` is the total number of tetrahedral cells in the grid, i.e., interior and dummy tetrahedral cells.

`nVert` is the number of vertices in the grid.

`nVertTot` is the total number of vertices in the grid, i.e., interior and dummy vertices.

`hexFlag` contains a flag indicating the kind of a given hexahedral cell. It is read in from the RocfluMP grid file, and can only take the values: `CELL_KIND_BNDRY`, `CELL_KIND_ACTUAL`, and `CELL_KIND_VIRTUAL` (defined in `ModParameters.F90`).

`hex2CellGlob` contains the mapping of a given hexahedral cell to a global cell.

`priFlag` contains a flag indicating the kind of a given prismatic cell. It is read in from the RocfluMP grid file, and can only take the values: `CELL_KIND_BNDRY`, `CELL_KIND_ACTUAL`, and `CELL_KIND_VIRTUAL` (defined in `ModParameters.F90`).

`pri2CellGlob` contains the mapping of a given prismatic cell to a global cell.

`pyrFlag` contains a flag indicating the kind of a given pyramidal cell. It is read in from the RocfluMP grid file, and can only take the values: `CELL_KIND_BNDRY`, `CELL_KIND_ACTUAL`, and `CELL_KIND_VIRTUAL` (defined in `ModParameters.F90`).

`pyr2CellGlob` contains the mapping of a given pyramidal cell to a global cell.

`tetFlag` contains a flag indicating the kind of a given tetrahedral cell. It is read in from the RocfluMP grid file, and can only take the values: `CELL_KIND_BNDRY`, `CELL_KIND_ACTUAL`, and `CELL_KIND_VIRTUAL` (defined in `ModParameters.F90`).

`tet2CellGlob` contains the mapping of a given tetrahedral cell to a global cell.

`vertFlag` contains a flag indicating the kind of a given vertex. It is read in from the RocfluMP grid file, and can only take the values `VERT_KIND_ACTUAL` and `VERT_KIND_VIRTUAL` (defined in `ModParameters.F90`).

`cellGlob2Loc` contains the mapping of a global cell to the local cell of a given type.

`c2cs` contains the cell stencils for each cell. This array is used in computing cell gradients and averaged variables.

`e2v` contains the two vertices defining an edge. This array is used only for grid motion.

`e2vTemp` is a temporary array used to construct `e2v`.

`f2c` contains the two cells adjacent to a face.

`f2cTemp` is a temporary array used to construct `f2c`.

`f2cs` contains the face stencils for each face. This array is used in computing face gradients.

`f2v` contains the vertices defining a face.

`f2vTemp` is a temporary array used to construct `f2v`.

`hex2v` contains the connectivity information for the hexahedral cells. The vertices must be numbered as shown in Fig. 6.6(a). The face to vertex, edge to vertex, and edge to face connectivity arrays for hexahedral cells are shown in Table 6.1.

`pri2v` contains the connectivity information for the prismatic cells. The vertices must be numbered as shown in Fig. 6.6(b). The face to vertex, edge to vertex, and edge to face connectivity arrays for prismatic cells are shown in Table 6.2.

`pyr2v` contains the connectivity information for the pyramidal cells. The vertices must be numbered as shown in Fig. 6.6(c). The face to vertex, edge to vertex, and edge to face connectivity arrays for pyramidal cells are shown in Table 6.3.

`tet2v` contains the connectivity information for the tetrahedral cells. The vertices must be numbered as shown in Fig. 6.6(d). The face to vertex, edge to vertex, and edge to face connectivity arrays for tetrahedral cells are shown in Table 6.4.

`fc` contains the face-centroid coordinates.

`fn` contains the components of the face-normal unit vector and the area of the face.

`cgwt` contains the cell-gradient weights.

`fgwt` contains the face-gradient weights.

`degr` contains the degree of each vertex.

`gs` contains the grid speed of each face.

`volMin` contains the minimum volume of all cells incident to a vertex. The variable is used in altering the effect of the grid-smoothing algorithm to avoid inverting cells and hence negative volumes.

`rhs` contains the residual for grid smoothing.

`xyz` contains the x-, y-, and z-coordinates of the vertices.

`vol` contains the volumes of the cells.

`cofg` contains the centroids of the cells.

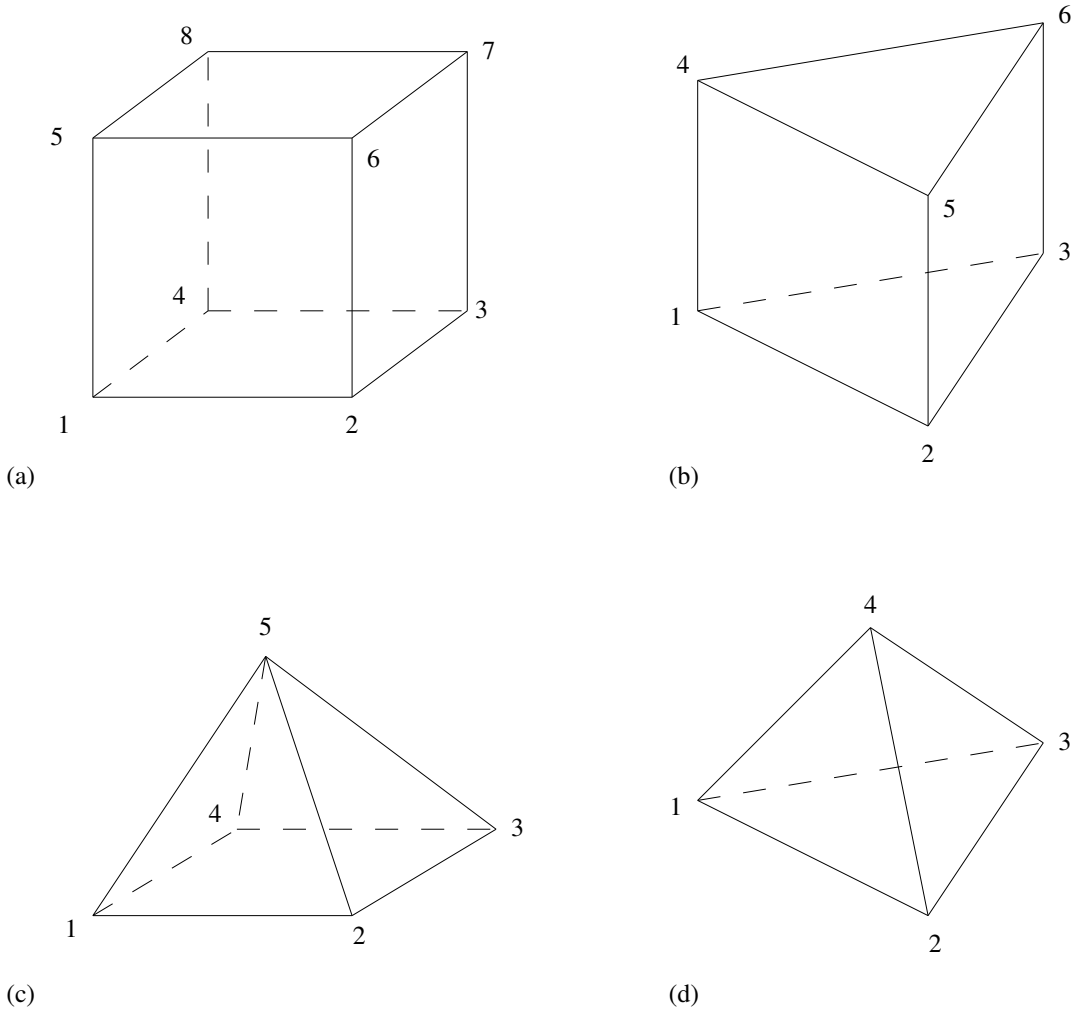

Figure 6.6: Definition of cell connectivity.

Face	Vertices			
1	1	4	3	2
2	1	2	6	5
3	2	3	7	6
4	3	4	8	7
5	1	5	8	4
6	5	6	7	8

Table 6.1: Face-to-vertex connectivity arrays for hexahedral cells.

Face	Vertices			
1	1	3	2	
2	1	2	5	4
3	2	3	6	5
4	1	4	6	3
5	4	5	6	

Table 6.2: Face-to-vertex connectivity arrays for prismatic cells.

Each cell type has not only a clearly defined numbering of its vertices, but also for its edges and faces. These numberings are listed in Tables 6.1-6.4. In reading these tables, it is to be understood that edges and faces have an orientation. This is a crucial point if the routines which construct data structures are to be understood properly. Therefore, the rows in these tables are to be read only from left to right. Thus, edge 10 of an hexahedron is pointing from vertex 2 to vertex 6. Furthermore, faces are oriented such that their normal vectors are pointing out of the cell. This corresponds to anti-clockwise ordering of the vertices when viewing the face of a cell from the outside of that cell, and to clockwise ordering when viewing the face through the cell.

Face	Vertices			
1	1	4	3	2
2	1	2	5	
3	2	3	5	
4	3	4	5	
5	1	5	4	

Table 6.3: Face-to-vertex connectivity arrays for pyramidal cells.

Face	Vertices		
1	1	2	3
2	2	4	3
3	1	3	4
4	1	4	2

Table 6.4: Face-to-vertex connectivity arrays for tetrahedral cells.

Figure 6.7: Overview of boundary data structure.

6.3.2 Module `RFLU_ModGrid.F90`

The module `RFLU_ModGrid.F90` contains some data structures used in the conversion of exterior grid formats to that used in RocfluMP, and some data structures used in the generation of other data structures.

6.4 Boundary Data Structure

An overview of the boundary data structure is given in Fig. 6.7. The definition of the boundary data structure is shown in Fig. 6.8.

The components of the user-defined type for the boundary data structure are defined as follows:

`bcType` is the type of the boundary patch. It is used to identify which boundary conditions is to be set on that boundary patch.

`bcCoupled` is a flag indicating whether the boundary patch is coupled to another code. It can have the values `BC_NOT_COUPLED`, `BC_NOT_BURNING`, and `BC_BURNING` (defined in `ModParameters.F90`).

`iPatchGlobal` is the global index of the boundary patch. For serial computations, `iPatchGlobal` is equal to the index of the boundary patch. The variable is needed to access the correct boundary condition information when reading the boundary condition file.

`nBFaces` is the total number of triangular and quadrilateral faces on a boundary patch.

`nBTris` is the number of triangular faces on a boundary patch, and is read from the grid file.

`nBQuads` is the number of quadrilateral faces on a boundary patch, and is read from the grid file.

`nBVert` is the number of vertices on a boundary patch.

```
  TYPE t_patch
    INTEGER :: bcType,bcCoupled
    INTEGER :: iPatchGlobal
    INTEGER :: nBFaces,nBTris,nBQuads,nBVert,nBVertEst
    INTEGER, DIMENSION(:), POINTER :: bf2bg,bf2c,bv,bvTemp
    INTEGER, DIMENSION(:,:), POINTER :: bf2cs,bf2ct,bf2v,bf2vl,bTri2v, &
                                        bTri2vl,bQuad2v,bQuad2vl
    LOGICAL :: movePatch,smoothGrid
    REAL(RFREAL), DIMENSION(:), POINTER :: gs
    REAL(RFREAL), DIMENSION(:,:), POINTER :: bvn,dXyz,fc,fn
    REAL(RFREAL), DIMENSION(:,:,:), POINTER :: bfgwt
    CHARACTER*(CHRLEN) :: bcName
#ifdef GENX
    INTEGER, DIMENSION(:), POINTER :: bFlag,bcFlag
    REAL(RFREAL), DIMENSION(:), POINTER :: mdotAlp,pf,qc,qr,rhofAlp,tempf, &
                                           tflmAlp
    REAL(RFREAL), DIMENSION(:,:), POINTER :: duAlp,nfAlp,rhofvfAlp,tracf, &
                                             xyz
#endif
    TYPE(t_bcvalues) :: valMixt,valTurb,valSpec,valPeul,valRadi
    TYPE(t_bcvalues), POINTER :: valPlag(:)
    TYPE(t_tile_plag) :: tilePlag
    TYPE(t_buffer_plag) :: bufferPlag
  END TYPE t_patch
```

Figure 6.8: Definition of boundary data structure.

`nBVertEst` is the estimated number of vertices on a boundary patch. It is used only in the construction of the boundary vertex list.

`bf2bg` is an access array which maps a face of a patch to an address in the array `bGradFace` contained in the mixture data type `t_mixt`. It is needed because the boundary face gradients of all boundary patches are stored in a single array for convenience.

`bf2ct` is an access array which maps a given boundary face to a cell type. It can only take the values `CELL_TYPE_TET`, `CELL_TYPE_HEX`, `CELL_TYPE_PRI`, and `CELL_TYPE_PYR` (defined in `ModParameters.F90`).

`bf2v` contains the vertices defining a boundary patch face.

`bf2vl` contains the vertices defining a boundary patch face, locally numbered for each boundary patch.

`bTri2v` contains the vertices defining a triangular boundary patch face. The vertices are oriented such that the normal vector is pointing out of the computational domain.

`bTri2vl` contains the vertices defining a triangular boundary patch face, locally numbered for each boundary patch. The vertices are oriented such that the normal vector is pointing out of the computational domain.

`bQuad2v` contains the vertices defining a quadrilateral boundary patch face. The vertices are oriented such that the normal vector is pointing out of the computational domain.

`bQuad2vl` contains the vertices defining a quadrilateral boundary patch face, locally numbered for each boundary patch. The vertices are oriented such that the normal vector is pointing out of the computational domain.

`movePatch` is a logical variable indicating whether the boundary patch is moving.

`smoothGrid` is a logical variable indicating whether the boundary patch grid is to be smoothed.

`gs` contains the grid speed of each boundary patch face.

`bvn` contains the components of the unit normal vector at the boundary patch vertices.

`dXyz` contains the imposed displacement of the boundary vertices.

`fc` contains the face-centroid coordinates.

`fn` contains the components of the face-normal unit vector and the area of the face.

`bfgwt` contains the face-gradient weights.

`bcName` contains the name of the boundary patch.

`bFlag` is a flag whether a burning face has ignited or not when running RocfluMP inside GENx. This is used to avoid faces which have ignited from extinguishing. (Used only if `GENX=1`.)

`bcFlag` is a flag indicating the type of interaction with other codes when running RocfluMP inside GENx. It can only assume the values `BC_NOT_COUPLED`, `BC_NOT_BURNING`, and `BC_BURNING` (defined in `ModParameters.F90`). (Used only if `GENX=1`.)

`mdotalp` contains the mass flux for each boundary patch face. It is allocated only for burning boundary patches. (Used only if `GENX=1`.)

`pf` contains the face pressure. (Used only if `GENX=1`.)

`qc` contains the convective heat flux. It is allocated only for burning boundary patches. (Used only if `GENX=1`.)

`qr` contains the radiative heat flux. It is allocated only for burning boundary patches. (Used only if `GENX=1`.)

`rhofalp` contains the fluid density for each boundary patch face. (Used only if `GENX=1`.)

`tempf` contains the fluid temperature for each boundary patch face. It is allocated only for burning boundary patches. (Used only if `GENX=1`.)

`tflmAlp` contains the static temperature of the injected fluid. It is allocated only for burning boundary patches. (Used only if `GENX=1`.)

`duAlp` contains the incremental displacement. (Used only if `GENX=1`.)

`nfAlp` contains the components of the unit face-normal vector. (Used only if `GENX=1`.)

`rhofvfAlp` contains the components of the product of the fluid density times the fluid velocity. Note that this is *not* the same as `mdotalp`, as the former also includes the effect of boundary motion due to deformation. (Used only if `GENX=1`.

`tracf` contains the fluid traction for each boundary patch face. (Used only if `GENX=1`.)

`xyz` contains the x-, y-, and z-coordinates of the vertices. (Used only if `GENX=1`.)

`valMixt` contains the user-specified values for the enforcement of boundary conditions on the mixture.

```
  TYPE t_mixt_input

    INTEGER :: flowModel
    LOGICAL :: moveGrid, externalBc
    INTEGER :: nDv, nTv, nGv, nGrad, indCp, indMol
    REAL(RFREAL) :: prLam, prTurb, scnLam, scnTurb

! - turbulence modeling

    INTEGER :: turbModel

! - species

    INTEGER :: specModel

! - continuum particles

    LOGICAL :: conPartUsed

! - discrete particles

    LOGICAL :: disPartUsed

! - radiation

    LOGICAL :: radiUsed

! - numerics

    INTEGER      :: spaceDiscr, spaceOrder, pSwitchType
    INTEGER      :: timeScheme, nrkSteps, ldiss(5)
    REAL(RFREAL) :: cfl, smoocf, vis2, vis4, pSwitchOmega, limfac, epsentr
    REAL(RFREAL) :: ark(5), grk(5), trk(5), betrk(5)

! - flow initialization (used within preprocessor)

    REAL(RFREAL) :: iniVelX, iniVelY, iniVelZ, iniPress, iniDens

! - flow initialization (for uniform flow preservation check)

    REAL(RFREAL) :: unifDens, unifEner, unifMomX, unifMomY, unifMomZ, unifPres

  END TYPE t_mixt_input
```

Figure 6.9: Definition of data type `t_mixt_input`.

6.5 Mixture Data Structure

6.5.1 Data Type t_mixt_input

flowModel is a flag indicating which flow model is used. It can only take the values FLOW_EULER or FLOW_NAVST (defined in ModParameters.F90).

moveGrid is a logical variable indicating whether the volume grid is to be moved. Note that the movement of interior points does not necessarily have to be activated when boundary patches are moving.

nDv contains the number of dependent variables. It is used to determine the size of the array dv (see below).

nTv contains the number of transport variables. It is used to determine the size of the array tv (see below).

nGv contains the number of gas variables. It is used to determine the size of the array gv (see below).

indCp is a flag used to allocate the array for specific heat in the gas-variable array. If the specific heat is to vary in space, indCp=1, otherwise indCp=0. This allows the gas-variable array gv (see below) to be accessed even if the specific heat is constant, which simplifies the code because conditional statements can be avoided.

indMol is a flag used to allocate the array for molar mass in the gas-variable array. If the molar mass is to vary in space, indMol=1, otherwise indMol=0. This allows the gas-variable array gv (see below) to be accessed even if the molar mass is constant, which simplifies the code because conditional statements can be avoided.

prLam contains the value of the laminar Prandtl number.

prTurb contains the value of the turbulent Prandtl number.

scnLam contains the value of the laminar Schmidt number.

scnTurb contains the value of the turbulent Schmidt number.

turbModel is a flag indicating which turbulence model is used.

specModel is a flag indicating which gas model is used. Currently, it can only take the value SPEC_MODEL_NONE.

conPartUsed is a logical variable indicating whether continuum particles are used.

disPartUsed is a logical variable indicating whether discrete particles are used.

radiUsed is a logical variable indicating whether radiation modeling is used.

`spaceDiscr` is a flag indicating which spatial discretization model is used. It can only take the values `DISCR_UPW_ROE` or `DISCR_OPT_LES`.

`spaceOrder` is a flag indicating the order of accuracy of the spatial discretization. Currently, it can only take the value `DISCR_ORDER_1`.

`nrkSteps` is a flag indicating the number of steps of the explicit-multistage or the Runge-Kutta scheme.

`ldiss` is a flag indicating whether the dissipation terms are to be computed in a given stage of the explicit multistage scheme.

`cfl` contains the value of the CFL number.

`epsentr` contains the value of the constant in the entropy fix.

`ark` contains coefficients used in the explicit-multistage and the Runge-Kutta scheme.

`grk` contains coefficients used in the Runge-Kutta scheme.

`trk` contains coefficients used in the Runge-Kutta scheme.

`betrk` contains coefficients used in the explicit-multistage scheme.

`iniVelX` contains the x-component of the velocity vector for the initial condition. It is only used in rfluprep and written into the solution file.

`iniVelY` contains the y-component of the velocity vector for the initial condition. It is only used in rfluprep and written into the solution file.

`iniVelZ` contains the z-component of the velocity vector for the initial condition. It is only used in rfluprep and written into the solution file.

`iniPress` contains the static pressure for the initial condition. It is only used in rfluprep and written into the solution file.

`iniDens` contains the density for the initial condition. It is used only in rfluprep and written into the solution file.

`unifDens` contains the density value when checking RocfluMP for uniform flow preservation. The check for uniform flow preservation is activated by compiling RocfluMP with `CHECK_UNIFLOW=1`.

`unifEner` contains the total internal energy value when checking RocfluMP for uniform flow preservation. The check for uniform flow preservation is activated by compiling RocfluMP with `CHECK_UNIFLOW=1`.

`unifMomX` contains the x-component of momentum when checking RocfluMP for uniform flow preservation. The check for uniform flow preservation is activated by compiling RocfluMP with `CHECK_UNIFLOW=1`.

`unifMomY` contains the y-component of momentum when checking RocfluMP for uniform flow preservation. The check for uniform flow preservation is activated by compiling RocfluMP with `CHECK_UNIFLOW=1`.

`unifMomZ` contains the z-component of momentum when checking RocfluMP for uniform flow preservation. The check for uniform flow preservation is activated by compiling RocfluMP with `CHECK_UNIFLOW=1`.

6.5.2 Data Type `t_mixt`

The data type `t_mixt` contains data related to the mixture and the solution of the associated transport equations. The variables associated with the mixture are divided into several types:

1. *Conserved variables*, i.e., dependent variables for which transport equations are solved, are stored in the array `cv`. For RocfluMP, the conserved variables are $\{\rho, \rho u, \rho v, \rho w, \rho E\}^t$.

2. *Dependent variables*, i.e., dependent variables for which no transport equations are solved, are stored in the array `dv`. For RocfluMP, the dependent variables are $\{p, T, c\}^t$.

3. *Transport variables*, i.e., dependent variables such as the coefficients of viscosity and conductivity.

4. *Gas variables*, i.e., dependent variables such as the specific heat at constant pressure and the molar mass.

The dependent, transport, and gas variables are updated after the update of the conserved variables by calling the routine `mixtureProperties.F90`.

Because it is convenient to work with different state variables at times, RocfluMP provides routines to change the "state" of the state vector from conserved variables to two different sets of primitive variables. This is advantageous when computing gradients for the viscous fluxes and printing information on the solution. The possible states are as follows:

1. Conserved variables given by $\{\rho, \rho u, \rho v, \rho w, \rho E\}^t$. This is the default state and indicated by `cvState` having the value `CV_MIXT_STATE_CONS`. The value of the integer parameter `CV_MIXT_STATE_CONS`, and the corresponding parameters for the other states, is defined in `ModParameters.F90`.

2. Primitive variables given by $\{\rho, u, v, w, p\}^t$. This state is indicated by `cvState` having the value `CV_MIXT_STATE_DUVWP`.

3. Primitive variables given by $\{\rho, u, v, w, T\}^t$. This state is indicated by `cvState` having the value `CV_MIXT_STATE_DUVWT`.

```
  TYPE t_mixt
    REAL(RFREAL), POINTER :: cv(:,:), cvOld(:,:), dv(:,:), tv(:,:), gv(:,:)
#ifdef STATS
    REAL(RFREAL), POINTER :: tav(:,:)
#endif
    REAL(RFREAL), POINTER :: rhs(:,:), rhsSum(:,:), diss(:,:), fterm(:,:)
    INTEGER :: cvState
    REAL(RFREAL), DIMENSION(:,:), POINTER :: cvVrtx
    REAL(RFREAL), DIMENSION(:,:,:), POINTER :: bGradFace,gradCell,gradFace
  END TYPE t_mixt
```

Figure 6.10: Definition of data type `t_mixt`.

Changes of the state are effected by `USE`ing the module `RFLU_ModConvertCv.F90`, and calling the routines:

`RFLU_ConvertCvCons2Prim(pRegion,cvStateFuture)` to convert from conserved variables to primitive variables. `cvStateFuture` must be set to either `CV_MIXT_STATE_DUVWP` or `CV_MIXT_STATE_DUVWT`; any other value will generate an error. An error will also be generated if `cvState` is not equal to `CV_MIXT_STATE_CONS`.

`RFLU_ConvertCvPrim2Cons(pRegion,cvStateFuture)` to convert from a primitive variable state to conserved variables. `cvStateFuture` must be set to `CV_MIXT_STATE_CONS`; any other value will generate an error. An error will also be generated if `cvState` is equal to `CV_MIXT_STATE_CONS`.

The strict checking of `cvState` upon calling the conversion routines is carried out to catch programming errors, where the state was changed on entering a routine, but not changed back on exiting the routine. Additional statements such as

```
IF ( pRegion%mixt%cvState == CV_MIXT_STATE_CONS ) THEN
  CALL ErrorStop(global,ERR_CV_STATE_INVALID,__LINE__)
END IF ! region
```

may be placed at the top of routines to catch such errors.

The data defined in the data type `t_mixt` is shown in Fig. 6.10, and explained in detail below.

`cv` contains the vector of conserved variables.

`cvOld` contains the vector of old conserved variables, i.e., from a previous timestep.

`dv` contains the vector of dependent variables.

`tv` contains the vector of transport variables.

`gv` contains the vector of gas variables.

`rhs` contains the residual vector.

`rhsSum` contains a weighted sum of residual vectors for the Runge-Kutta scheme.

`diss` contains the residual vector due to dissipative terms of the spatial discretization.

`cvState` is a flag indicating which state is stored in the conservative state vector. It can only take the values `CV_MIXT_STATE_CONS`, `CV_MIXT_STATE_DUVWP`, and `CV_MIXT_STATE_DUVWT`.

`cvVrtx` contains the state vector at the vertices. It is used only in rflupost after having interpolated the cell-centered values to the vertices.

`bGradFace` contains the boundary-face gradients. It is accessed using the array `bf2bg` in the data type `t_patch`.

`gradCell` contains the cell gradients.

`gradFace` contains the face gradients.

Chapter 7

Parallel Implementation

Chapter 8

GENx Integration

Chapter 9

Installation and Compilation

9.1 Installation

The following assumes that RocfluMP is to be installed either from the CSAR CVS repository or from a `gzip`ped `tar` file.

9.1.1 Installation from CVS Repository

To be able to access the CSAR CVS repository, set the `CVSROOT` environment variable to (taking the `bash` shell as an example)

```
export CVSROOT=:pserver:user@machine.uiuc.edu:/cvsroot
```

and either open a new terminal or type

```
[user@machine ~]$ source .bashrc
```

Then type

```
[user@machine ~]$ cvs login
```

and hit the `Enter` key at the prompt.

Now move into the directory where you want to install RocfluMP. In the following, this is assumed to be `directory`. Then type

```
[user@machine ~/directory]$ cvs co genx/Codes/RocfluidMP
```

which will check out the source code for RocfluMP from the repository.

Assuming the checkout command has completed successfully, you are now ready to compile the code for serial computations, and you can proceed to Sec. 9.2.

9.1.2 Installation from `.tar.gz` File

Move into the directory where you want to install RocfluMP. In the following, this is assumed to be `directory`. Move or copy the `gzipped` `tar` file, assumed to be `<file>.tar.gz` in the following, into `directory`. Then type

```
[user@machine ~/directory]$ gzip -d <file>.tar.gz
[user@machine ~/directory]$ tar -xvf <file>.tar
```

which will unpack the source code.

Assuming these commands to have completed successfully, you are now ready to compile the code for serial computations, and you can proceed to Sec. 9.2.

9.2 Compilation

9.2.1 Overview of Compilation Process

The compilation process for RocfluMP is automatic in the sense that the `Makefiles` determine the machine type and set the suitable compilation options. If you intend to run on Apple, IBM, Linux, SGI, or Sun machines, you do not need to modify any `Makefiles`. If you intend to run on other machines, you will need to create your own `Makefile`. You can pattern it after the existing machine-dependent `Makefiles`.

RocfluMP is compiled with MPI by default, which means that you must have installed MPI on your machine before attempting to compile RocfluMP.

The compilation process consists of two stages. The first stage is the actual computation, as described below. The output of the compilation process are several executables:

rfluconv The conversion module of RocfluMP.

rfluinit The initialization module of RocfluMP.

rflumap The region mapping module of RocfluMP.

rflupick The region and cell picking module of RocfluMP.

rflupost The postprocessing module of RocfluMP.

rflupart The partitioning module of RocfluMP.

rflump The flow solution module of RocfluMP.

The second stage consists of copying these executables into your `$(HOME)/bin` directory by typing

```
[user@machine ~/directory]$ gmake RFLU=1 install
```

9.2.2 Description of Compilation Options

To compile RocfluMP, type the following at the prompt:

```
[user@machine ~/directory]$ gmake RFLU=1 <options>
```

where the currently supported `<options>` are any of the following:

`CHECK_DATASTRUCT=1` Activates checking of data structures. This option will print out the content of the important data structures used by RocfluMP. Note that activating this option will lead to substantial screen output, so it should only be activated for small cases.

`DEBUG=1` Activates debugging compiler options. If this option is not specified, optimizing compiler options are chosen by default.

`PLAG=1` Activates compilation of Rocpart. This option must be specified if you wish to run computations with Lagrangian particles.

`SPEC=1` Activates compilation of Rocspecies. This option must be specified if you wish to run computations with chemical species and/or Equilibrium Eulerian particles.

References

[1] Bruner C.W.S., "Geometric Properties of Arbitrary Polyhedra in Terms of Face Geometry", AIAA J., 33(7):1350, 1995.

[2] Wang Z.J., "Improved Formulation for Geometric Properties of Arbitrary Polyhedra", AIAA J., 37(10):1326, 1999.