Computational Fluid Dynamics (CFD) with General Equation Mesh Solver (GEMS)

by Xuxiao Li

May 3, 2020

Contents

Prologue

In this document, I will cover multiple topics on both the theoretical and practical aspects of computational fluid dynamics (CFD). With such a rich range of contents in CFD, I will only focus on certain methods that I am specialized in, while general concepts will only be briefly discussed. The major method discussed in this document will be the Finite Volume Method (FVM). FVM is a somewhat "old school" method compared to the more recent and advanced variants of the Finite Element Method (FEM). However, each method has its own advantages and disadvantages when applied to specific problems. FVM served for a long period of time as the workhorse in solving the fluid mechanics problems, while FEM was born to solve solid mechanics problems. This is a big gap between the "fluid" and the "solid" community. The trend is to merge the merits from both methods (e.g., discontinuous Galerkin), but that is still an ongoing research topic and indeed, old habits die hard.

Within the "fluid" community, there is a division between those studying compressible flow problems and those studying incompressible flow problems. Accordingly, the CFD methods can be divided into the density-based methods (suitable for compressible flow) and pressure-based methods (suitable for incompressible flow). The major difference here is that the fluid velocity in compressible flow is typically large and comparable to the sound speed, while fluid velocity is relative small for incompressible flows (small Mach number). This gap has already been filled through years of efforts. Specifically, density-based methods can also solve incompressible flow problems using the preconditioning technique, which gives a unified framework for solving fluid dynamics problems. This approach will be the focus of this document.

In the 80's, there were several pioneers who contributed significantly to the preconditioning methods, e.g., Eli Turkel, Bram Van Leer, and Charles Merckle. This document will be focused on Merckle's preconditioning system, and in fact, the practical part of the document is made based on one of the in-house codes developed at Merckle's research group. The in-house code is named as the General Equation Mesh Solver (GEMS) whose main creator is Dr. Ding Li. He worked as a research associate with Prof. Merckle, initially at the University of Tennessee, and later at Purdue University. Dr. Li embarked on the development of GEMS about early 1999. In 2002, the version 1.0 is completed. In 2005, he has added the Maxwell equation into GEMS and also refined multiple features to improve the generality of the code. In a paper Dr. Li published in 2006, he demonstrated the capability of GEMS with impressing results.

I feel obliged to mention how I can have the access to the GEMS code. During the time

Dr. Ding Li was at Purdue university, there was a PhD student named Shaoyi Wen who worked with Dr. Li. Shaoyi was then advised by Prof. Yung Shin whose research group had some collaborations with Prof. Merckle's group. Shaoyi modified GEMS code for his needs with the help of Dr. Li and published a paper in 2010 using GEMS to solve thermal-fluid problems in direct laser deposition processes. Thereafter, the GEMS code seemed to be made available to Prof. Shin's group. Before Shaoyi graduated from Shin's group, he passed the GEMS code to another PhD student at Shin's group, Wenda Tan. At this time, Dr. Li has left Purdue (I actually don't know where he went). Wenda has never made acquaintance with Dr. Li. However, he managed to exploit the GEMS code and have three papers published in 2013, 2014 and 2015 with it. In these papers, Wenda simulated the laser keyhole welding processes, and with GEMS, his model incorporated multiple physics and has high fidelity. In 2015, Wenda ceremoniously graduated from Purdue and became an assistant professor at the University of Utah. In the fall of 2015, I was registered as a PhD student at the University of Utah and I was advised by Wenda (Dr. Tan) since then. Therefore, I have the privilege to study the GEMS code and apply it for my PhD research. I have been exploring the GEMS code since 2017 and still learn new things about it today. So far I only touched upon the "basic" functions in GEMS which is to solve the basic equations: mass, momentum, energy, and species conservation equations. The unexplored part is: turbulence modeling and incorporation of Maxwell equations. In the definitive version of the GEMS, all the equations could be solved and can be customized such that multiple sets of equations can be assigned to different partitions of the domain. In this document, I will focus on the "basic" package which I feel the most comfortable with.

The philosophy of this document will be that more focus is put on the theoretical side. By that I do not mean that I will linger on derivations of equations, as "computational" fluid dynamics should emphasize on **data structure** and **algorithm** and not on mathematical reasoning. However, I will not go into details such as "this subroutine does this", or "this variable is used for this", or "how do I output such information". Instead, I will try to **speak out** what the code is trying to do, why it should be doing this, and how certain things are done. With this global picture, you should be able to "code it up" by any language, any way you want. In situations where practical guide is necessary, I will try to separate it by distinct sections.

Chapter 1: Mesh

1.1 Definition of Basic Mesh

The mesh describes the data structure of carrying out CFD simulations. Mesh is a discretization of a calculation domain (a physical domain in which CFD simulations is conducted). For example, we have a two-dimensional rectangle in which we want to carry out CFD simulations, as shown in Fig. 1.1a. An exemplary mesh is shown in Fig. 1.1b. The rectangle has a width of 2 of a height of 1. The boundary conditions are defined as follows. The left and right edge form boundary condition batch 1, the top edge is boundary condition batch 2, and the bottom edge is boundary batch 3.

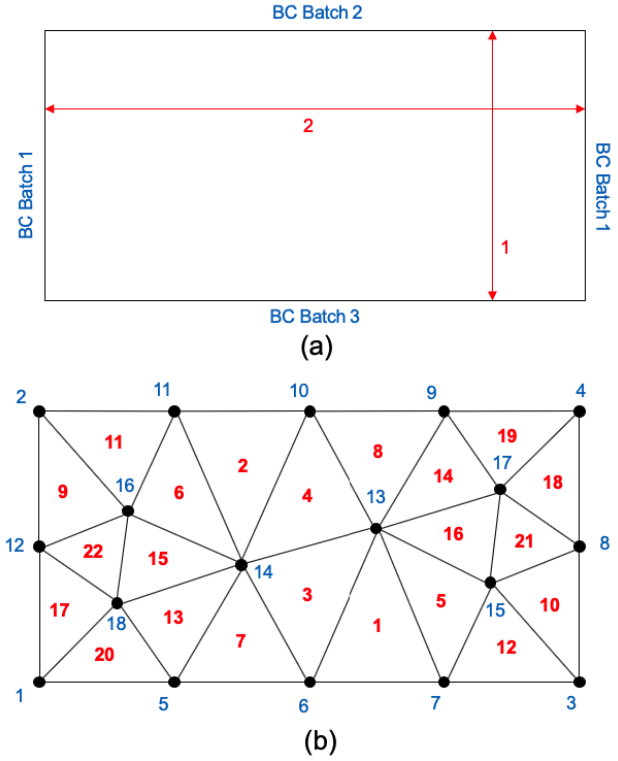

Figure 1.1: Illustration of mesh. (a) Definition of the calculation domain. (b) A discretization of the calculation domain (mesh).

Several notes on defining the calculation domain. First, the dimensions are be scaled up or down when simulating a particular problem. That is, the numbers of 1 and 2 can mean

"meter" or "micrometer" depending on the specific problem. No need to specify units here. Second, the boundary conditions must be carefully stated. We need to first think about how many distinct BC's we want to define. Then we assign each distinct BC (BC batch) to a segment of the domain boundary. Here, we have three unique BC's. Each of them is assigned with a certain segment of the domain boundary.

After the calculation domain is defined. We seed the domain with a set of points (black circles in Fig. 1.1b) in the domain, referred to as the "**nodes**". The nodes are indexed by the blue numbers. We have 18 nodes in the domain. Next, we connect the nodes to form a "tessellation" of the domain. Here we use all triangles. In general, there is a large flexibility of the tessellation. You can use triangles, polygons, or their mixture in two-dimension. In three-dimension, there are the tetrahedral, pyramid, prism, and hexagon, or their mixture (called hybrid mesh). This flexibility is incorporated in the GEMS code, therefore the name "General Equation Mesh Solver". Now with the tessellation, we divide the domain into triangular "elements", from which the name "Finite Element Method" is originated. However, we will only focus on the Finite Volume Method in this document, and the elements will be referred to as the "**cells**". We have 22 cells in the domain and they are indexed with the red numbers. It should be mentioned that the indexing order of the nodes and cells does not matter in the definition of the mesh.

Now we can define the "**cell-node connectivity**" by associating cells with its corresponding nodes. For example:

```
cell 1 ---> node 6, 7, 13
cell 2 ---> node 11, 10, 14
...
```

With that, let's design the following data structures `cell` and `node` to better represent this information. For `node`, we have (in Fortran):

```
type node
   real :: xyz(ndim)
end type node
```

where `ndim` is the number of dimensions and `xyz` is the coordinate of the node. We will need to have an array of nodes `nodes(:)` to store all the information about nodes. For cells, we have the following data type:

```
type cell
   real :: centp(ndim)
   integer, pointer :: c2n(:)
end type cell
```

where `centp` is the centroid of the cell and `c2n` is the node indices associated with this cell. Note that we can calculate the centroid of the cell based on the nodal coordinates associated with the cell. We will also need an array of cells `cells(:)` to store all the information about cells. Now we have introduced two important data types `node` and `cell`. Each of these data types can have more attributes which we will build on later.

Now we still need to add the last part of the mesh definition, the boundary conditions. To define the boundary conditions of the mesh, we have to introduce another important concept, "**faces**". Faces are the edges (or facets in 3D) that form the boundary of a cell. There are three faces per one triangle cell, and six faces per one hexagon cell (in 3D), etc. Faces that are shared between two cells are defined as **interior faces**. Faces that are only belong to a single cell are defined as **boundary faces**. Faces can be distinguished by a set of nodes, which brings about the "face-node" connectivity. We will discuss in further details about faces in the next section. For now, we use face to define the boundary conditions as follows:

```
BC batch 1 ---> 4 faces: (1, 12), (2, 12), (3, 6), (4, 8)
BC batch 2 ---> 4 faces: (2, 11), (11, 10), (10, 9), (9, 4)
BC batch 3 ---> 4 faces: (1, 5), (5, 6), (6, 7), (7, 3)
```

That is, we use the boundary faces (shown as node sets above) to define the segments of boundaries. Each set of faces represents a unique BC. It is convenient to create a data type for the BC's:

```
type bc_type
   integer :: label
   integer :: igrp
   integer :: itype
end type bc_type
```

Here, `label` is a distinct integer to indicate the batch of BC. `igrp` indicates which "group" of BC it is. In GEMS, there are 6 groups of BC's:

- Group 1, Inlet
- Group 2, Outlet
- Group 3, Farfield
- Group 4, Wall
- Group 5, Geometric (e.g., symmetric, periodic)
- Group 6, MHD (for Maxwell equations)

Each group of BC can have sub-categories (types) which is saved in the attribute `itype`. Again, we need an array of BC's `bc(:)` to store all the information. Apparently, the association between BC and faces should be defined based on the face lists of each batch of BC, which will be discussed in the next section.

So far, we have introduced the complete information to define a mesh, summarized as follows:

1. Nodal coordinates.
2. Cell-node connectivity.
3. Face list in each BC batch.

We emphasize this contains the complete information of a mesh. We refer this piece of information as the "basic mesh" or the finite element mesh. The information is complete but is not organized in a way convenient for finite volume method. As we will see, additional

data types can be constructed based on the basic mesh and be used to facilitate finite-volume computation.

1.2 Finite Volume Mesh

We establish the data type `face` from the basic mesh to form the "finite volume mesh". The `face` type has much more emphasis in the finite volume method as the fluxes on the faces need to be computed. On the opposite side, the finite element method emphasize on the data type `node`. The `face` type is defined as:

```
type face
   integer :: itype
   type(cell), pointer :: left_cell
   type(cell), pointer :: right_cell
   integer, pointer :: f2n(:)
   real :: centp(ndim)
end type face
```

Here, `itype` defines the type of the face which we will elaborate later. The pointers `left_cell` and `right_cell` refers to the two cells that share this face. This is referred to as the **face-cell connectivity**. The pointer `f2n(:)`, like `c2n(:)`, is the **face-node connectivity**, which refers to the set of nodes that defines the face. At last, the center of the face `centp` can be calculated based on the nodal coordinates which can be found through `f2n(:)`. The `face` type constructed based on Fig. 1.1b is shown in Fig. 1.2. The faces are indexed with the green number and there are 39 faces in the calculation domain.

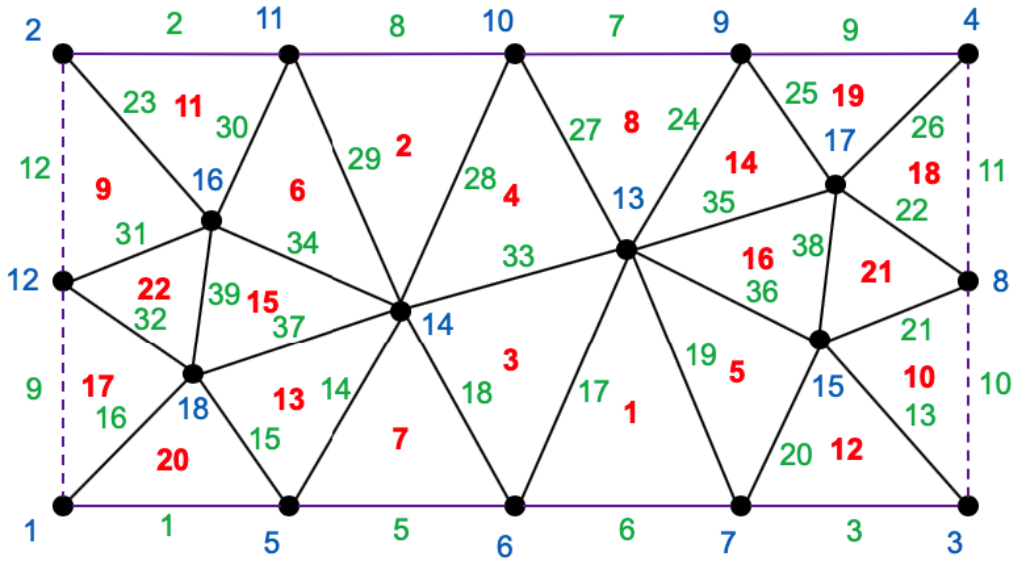

Figure 1.2: Construction of faces to make the finite volume mesh.

How to collect all the face information from the basic mesh? The face information can be found from the cell-node connectivity. For example:

```
cell 1 ---> node 6, 7, 13, we can find:
```

```
    face 1 --> node 6, 7
    face 2 --> node 7, 13
    face 3 --> node 13, 6
cell 2 ---> node 11, 10, 14, we can find:
    face 4 --> node 11, 10
    ...
```

By doing so we obtain an array of faces, `faces(:)`. Two points need to be noted. First, the face-node connectivity is contained in the cell-node connectivity, but we still need to know which nodes in a cell can form a face. If all cells are triangles, any combination of the three nodes of the cell will form a face. But for quadrilaterals and 3D cells, a set of rules need to be pre-defined to help extracting the face-node connectivity. The second note is, faces identified this way will be double-counted. We need to delete the repeating faces. This can be done by checking whether the face-node connectivity `f2n` is repeating. In doing so, every time we find a repeating face, it means two cells share the same face. Therefore, the `left_cell` and the `right_cell` can be found. We note that the "left" and "right" do not mean the literal directions. In practice, we always make the cell with the smaller index (red number in Fig. 1.2) to be the left cell and the other cell to be the right cell.

There are some faces that are only belong to a single cell. These faces are referred to as the "**boundary faces**" (face 1 - 12 in Fig. 1.2). Boundary faces are regarded as special faces and we will make them to be at the front of `faces(:)` by **swapping faces** in the face array. We assign the only cell that a boundary face is associated to be the **left cell** of the boundary face. That is, the boundary faces only have `left_cell` which is the interior cell, for now.

In CFD computation, it is required that every face be associated with two cells, including the boundary faces. Therefore, we need to create a "**ghost cell**" as the right cell for the boundary faces. Ghost is an overused word (but I still sometimes use it). The ghost cell is alternatively referred to as the "**boundary cell**". To create the ghost cells, we can simply mirror the interior cell with respect to the boundary face. However, there is one exception, the periodic boundary condition. For periodic BC, we need to translate the same cell "on the other side" to form the ghost cell. To illustrate, the boundary faces are marked purple in Fig. 1.2. We let BC batch 1 to be the periodic BC, and the periodic boundary faces are marked as dashed purple. To create the boundary cells, we can mirror cell 11, 2, 8, 9, 20, 7, 1, and 12 for face 2, 8, 7, 9, 1, 5, 6, 3, respectively. For periodic boundary cells, we need to translate cell 9, 17, 18, 10 for face 11, 10, 12, 9, respectively. This indicates that the periodic boundary faces must "match", otherwise, the mesh is ill-defined for the periodic BC. The periodic boundary faces are special boundary faces. We will move these faces to the end of the boundary faces. That is, `face(1)` to `face(12)` are the boundary faces, and `face(9)` to `face(12)` are reserved for periodic boundary faces. We will dedicate an entire section to periodic BC later.

One last note about face types. The interior faces (black solid line in Fig. 1.2) has `itype = 0`. The boundary faces has a positive `itype` which equals to the index in `bc(:)` that the boundary face is associated with. For example, we have 3 batches of BC's, and we create an array `bc(:)` with a length of 3. Each element in `bc` corresponds to one specific BC batch.

Say, batch 1 corresponds to `bc(3)`. Then the `itype` for face 9, 10, 11, 12 will be equal to 3. There can be other `itypes` which we shall discuss later.

To summarize, the finite volume mesh consists of the following information:

1. Nodal coordinates.
2. Cell-node connectivity.
3. Face types, face-cell connectivity and face-node connectivity.

Notice that the information about BC's can be fully described by the `bc(:)` array and the `itype` in `face`.

1.3 Partitioning of Mesh

Now we have got a finite volume mesh, with three essential data types, `node`, `cell`, and `face`. It is no problem to plug this mesh into some CFD code. However, today's computation lives in a parallel world. Any practical CFD code must enable parallel computing. GEMS uses MPI for parallel computing. In MPI, the finite volume mesh is decomposed into partitions. Each partition is assigned to one "core" (or one CPU, one "processing element", any way you want to call it). Each core only has information about its own partition of the entire domain and has no idea of what the entire domain is like. Such a partitioning is shown in Fig. 1.3.

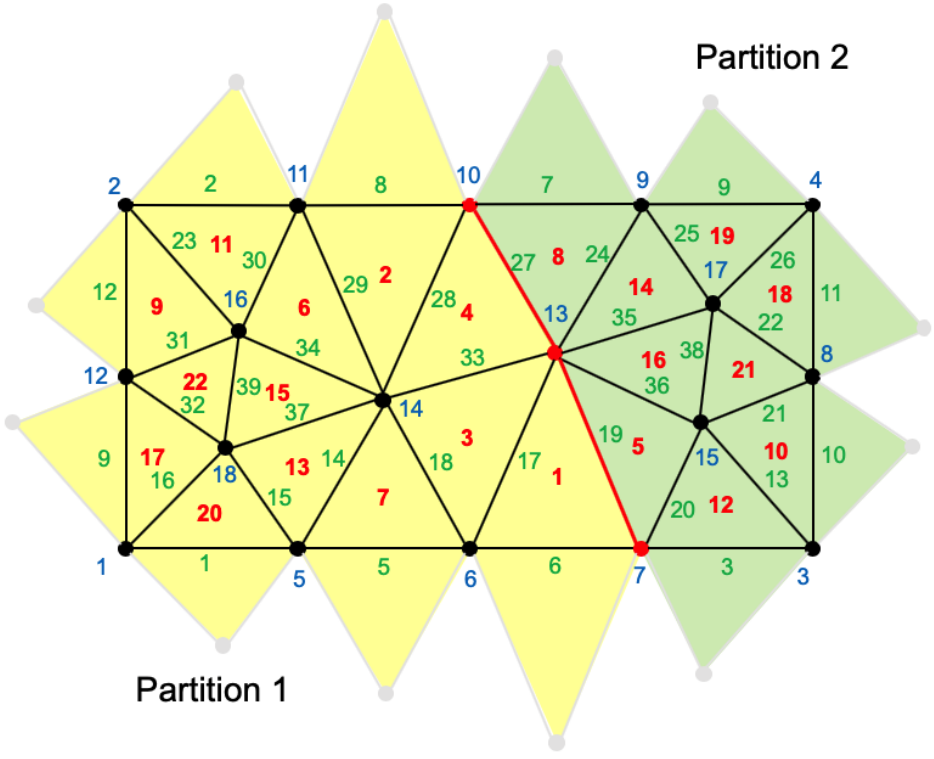

Figure 1.3: Partitioning of calculation domain

Here the contour formed by the solid black line is the calculation domain, including the

nodes (indexed by blue), cells (indexed by red) and faces (indexed by green). We also show the ghost cells (boundary cells) marked by the gray lines. It is noted that the ghost cells can be entirely derived from the boundary faces and need not be store prior to CFD simulation. Based on this configuration, we partition the domain into two partitions. First, we identify which **cells** belong to which partition. Then the nodes and faces shared by cells from both partitions can be identified (marked by red). These delineates the **interface** of the partitions. The nodes, cells, and faces belong to partition 1 is shadowed by yellow and those belong to partition 2 is shadowed by green.

Now we split the domain into two partitions as shown in Fig. 1.4. Take partition 1 for example, the nodes, cells, and faces all need to be re-indexed as partition 1 does not have any information about partition 2. The re-indexed numbers are shown by the corresponding blue, red and green numbers. For partition 1, to establish the "communication" with the external world (which means partition 2 in this case), we identify all the cells (including ghost cells) in partition 2 that shared nodes with partition 1. Those are the orange-shaded cells indexed from 1 - 7. This means there are 7 cells we need to create in partition 1 as "containers" to receive the information from partition 2. Same for partition 2, we need to create 6 "container" cells to receive information from partition 1. From a global point of view (Fig. 1.3), the container cells (orange-shaded cells in Fig. 1.4) for both partitions can be identified. Then, we can correspondingly mark in each partitions the cells that need to be "sent" out to the external world (the sky-blue shaded cells in Fig. 1.4). Note that the "sending cells" and the "receiving cells" must match. That is, the receiving cells in partition 1 (orange) must be match the sending cells in partition 2 (sky-blue), and vice versa.

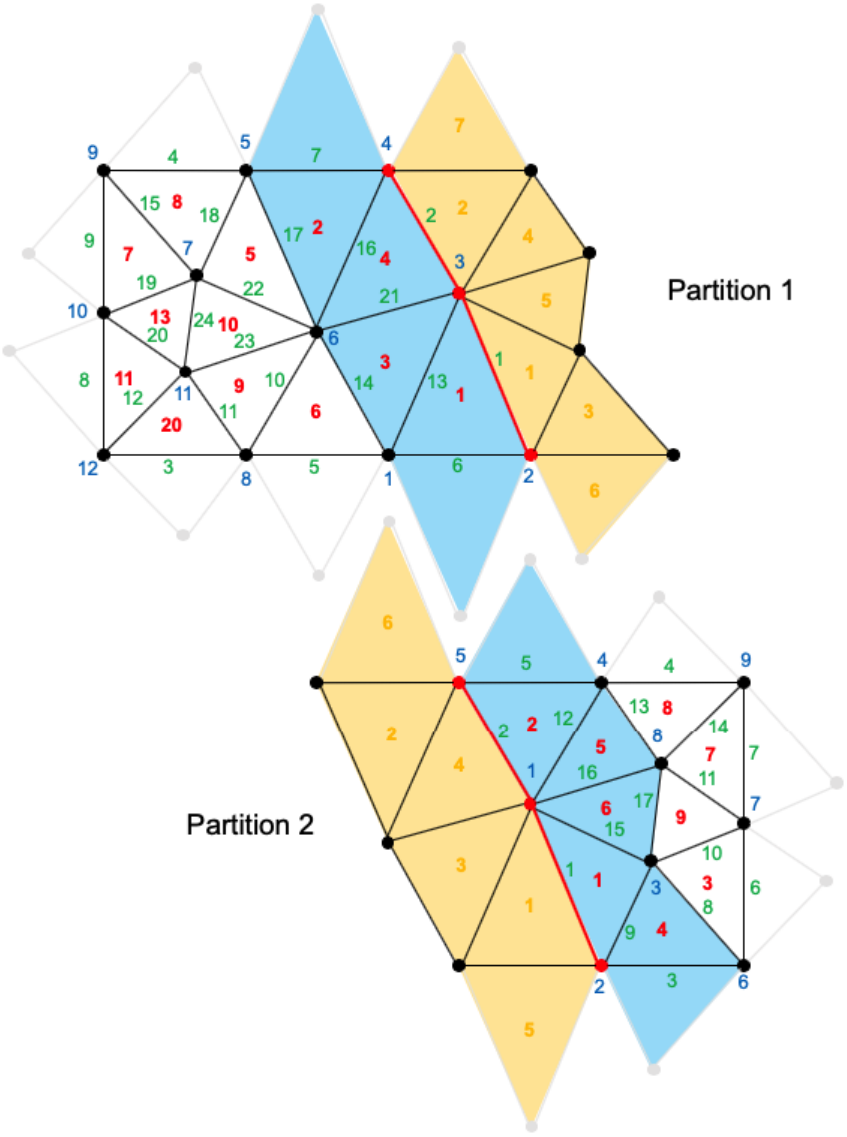

Figure 1.4: Splitting of the domain and re-indexing.

To represent the data structure for communication among partitions, in GEMS, the data type `itf` (standing for "interface") is created:

```
type itf
   integer :: nn
   integer, pointer :: np(:), nnp(:)
   type(cell), pointer :: pcell(:)
end itf
```

where `nn` is the number of partitions from which the current partition receives data; `np` and `nnp` are arrays with a length of `nn`. `np` stores the id of the partitions from which the current partition receives data, and `nnp` stores the number of cells to receive from each partition. Finally, `pcell` is the array of "container" cells to store the received data. `pcell(:)` should have a length of `sum(nnp)`. It should be noted that the `pcell(:)` array needs to be **allocated**

as it is ghost cells that do not exist in the physical domain of partition 1.

So far, the attributes of `itf` are only relative to receiving (orange-shaded cells in Fig. 1.4). We need to add similar attributes for sending:

```
type itf
   ...
   integer :: nc
   integer :: cp(:), ncp(:)
   type(neighbour_cell), pointer :: scell(:)
end type itf
```

Here `nc` is the number to partitions to which the current partition needs to send data; `cp` is the id's of partitions to which the current partition sends data and `ncp` is the number of cells to send to each partition in `cp`. The array `scell(:)` points to the cells in the current partition whose information will be sent to external partitions. Notice that instead of using `type(cell)`, we invented a new data type `neighbour_cell` for `scell(:)`. This data type is a **nested type** for the type `cell`:

```
type neighbour_cell
   type(cell), pointer :: to_cell
end type neighbour_cell
```

There is nothing in the type `neighbour_cell` but a `cell` pointer. This nested structure is adopted in GEMS to represent that the cell pointer **needs not be allocated**. Rather, it is merely a reference to cells already allocated.

How can we construct such data type `itf`? It must start from the global domain (Fig. 1.3). First we identify the container cells for each partition (1 & 2). Then we can identify the sending cells in partition 1 based on the container cell in partition 2, and we record the cell id's and boundary face id's (for boundary cells to send). Same for partition 2. For container cells, they don't yet exist in each partition. Therefore, (for example, in partition 1), we simply allocate 7 "empty" cells. The empty cells will be fulfilled once the data is received from partition 2.

Although the container cells are empty, the cell-node and cell-face connectivity between the container cells and the nodes and faces needs to be identified. Take partition 1 for example, for cell-node connectivity, we need to record in each container cell which node index in partition 1 it is associated with. Note, only record those nodes in partition 1, so we have 2 nodes for `pcell(1:2)` and only 1 node for `pcell(3:7)`. For face-cell connectivity, we need to associate the `right_cell` of `faces(1:2)` with `pcell(1:2)` for partition 1. For partition 2, we associate the `right_cell` of `faces(1:2)` with `pcell(1)` and `pcell(4)`. The `left_cell` will be assigned to the other cell which is interior cell. It should be mentioned that the `left_cell` of a face always belong to the interior of the domain.

The indexing convention for the partitioning is as follows. We first index interior cells for `pcell` and then the boundary cells. Same for the `scell(:)`. Note, the indexing of `scell(:)` and `pcell(:)` must match across partitions. The faces that has a `right_cell` as a container

cell are referred to as the "**partitioning faces**" and are considered as another type of special faces. A special (large) number is reserved for the `itype` of the partitioning faces, 19621011. The partitioning faces are indexed from very beginning of the `faces` array (by proper face swapping). Therefore, we have a rather complex rule for face indexing (take partition 1 as example):

1. Index partitioning faces (face 1 & 2), `itype = 19621011`
2. Index boundary faces that are not periodic boundaries (face 3 - 7), `itype =` positive number (from one to the number of BC's)
3. Index periodic boundary faces (face 8 & 9), `itype =` the number for periodic BC
4. Index interior faces (face 10 - 23) , `itype = 0`

Also, we record the number of partitioning faces in the type `itf` by adding the variable `nitf`:

```
type itf
   ...
   integer :: nitf
end type itf
```

With the mesh partitioning, we will generate separate mesh files for each partition. Let us now summarize the information contained in each mesh file:

1. Nodal coordinates for current partition
2. Cell-node connectivity for current partition
3. Face-cell connectivity, face-node connectivity as well as face type for current partition
4. Sending information: No. of sending-to partitions, No. of total sending cells. For each sending-to partition,
   (a) sending-to partition id, No. of to-be-sent interior cells, No. of to-be-sent boundary cells
   (b) for to-be-sent interior cells, store those cell id
   (c) for to-be-sent boundary cells, store those boundary face id
5. Receiving information: No. of receive-from partitions, No. of total receiving (container) cells. For each receive-from partition,
   (a) receive-from partition id, No. of container cells to receive from this partition
   (b) for each container cell, record the node shared between container cell and the current partition

1.4 Periodic Boundary Condition

The treatment of periodic BC deserves a separate section to describe. The difficulty in treating periodic BC arises when information in the boundary cells for periodic BC cannot be directly obtained from the left cell of the boundary face (i.e., mirroring). Rather, we need

to find the left cell of the matching boundary face "on the other side". Moreover, when the domain is partitioned, the matching face may not even exist in the current partition, and communication must be established specially for treating periodic BC.

To better illustrate the treatment of periodic BC, we modified the original definition of BC's in Fig. 1.1. As shown in Fig. 1.5a, we define only two batches of BC's and both of them are assigned to be periodic BC's. Each batch consists of a pair of walls. The corresponding mesh (before partition) is shown in Fig. 1.5b. We emphasize again that boundary faces in the mesh for periodic BC must "match", i.e., sharing the exact same nodal coordinates in all directions excluding the periodic direction. Otherwise, the face meshing is ill-defined for the periodic BC.

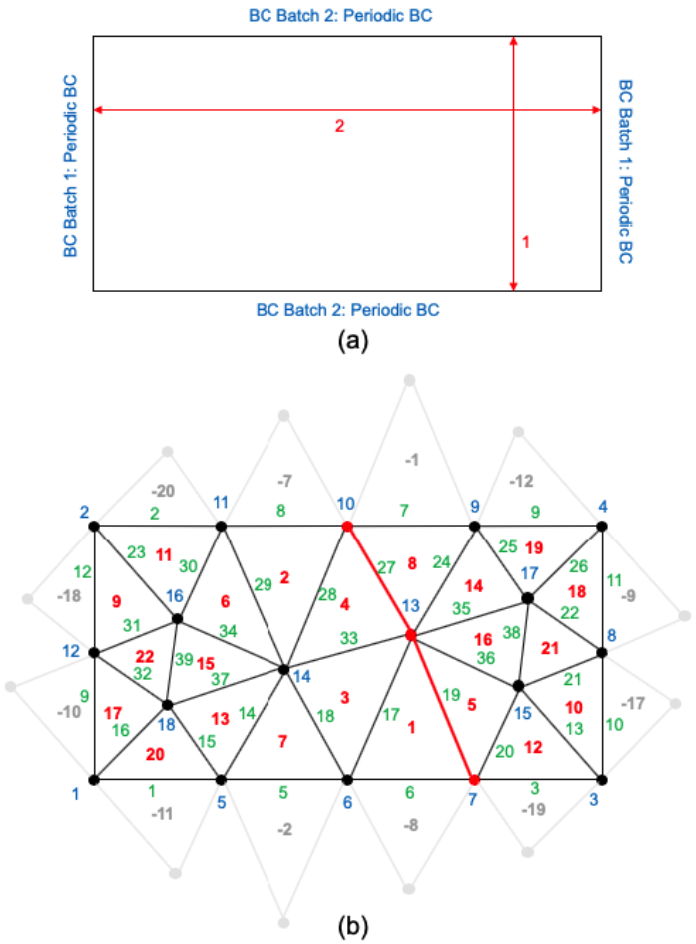

Figure 1.5: Re-define the boundary conditions. (a) Set two pairs of walls as 2 batches of periodic BC. (b) The mesh corresponding to the setup in (a).

In Fig. 1.5b, again, the node, cells, and faces of the global domain are indexed. The global domain will be partitioned into two parts as separated by the red line like before. The boundary cells are marked by the gray edges. The first step to construct periodic BC is to find the "matching cells" for the boundary cells. This step is shown with the global indices in Fig. 1.5b. We index the periodic boundary cells by a negative number whose absolute

value matches the global indices of the matching interior cells, as shown in Fig. 1.5b. After the matching cells are found (globally), we merge the two (or more) batches of periodic BC as one batch. This is due to the special treatment of periodic BC in GEMS. As long as matching cells are found, the complete information regarding to all periodic BC's is known. Therefore, they are viewed as a single package (of BC) in GEMS.

Now, let's illustrate the specific data structure of periodic BC by considering the partitioning layout shown in Fig. 1.6. Take partition 1 for example, the yellow region is the physical domain of partition 1 and the orange region is the container cells to receive information from partition 2, like before. There are 7 container cells, as indexed by the orange numbers. Now, the periodic boundary cells are considered as a second type of container cells, as indexed by the purple numbers. There are 8 **periodic container cells** and they are indexed as negative number to differentiate with the "partitioning container cells". It is noted that some of these periodic container cells needs to be received from partition 2 (-4, -6 and -7), but some of them are actually just the interior cells in partition 1 but "on the other side" of the boundary faces of -2, -5, -1, -3. GEMS will treat all the periodic container/boundary cells by communications between partitions, in a unified manner. For cell -2, -5, -1 and -3, they will be communicated from partition 1 and to partition 1. This is the difference between communicating partitioning cells and periodic boundary cells. The latter involves communicating with the partition itself (totally OK in MPI).

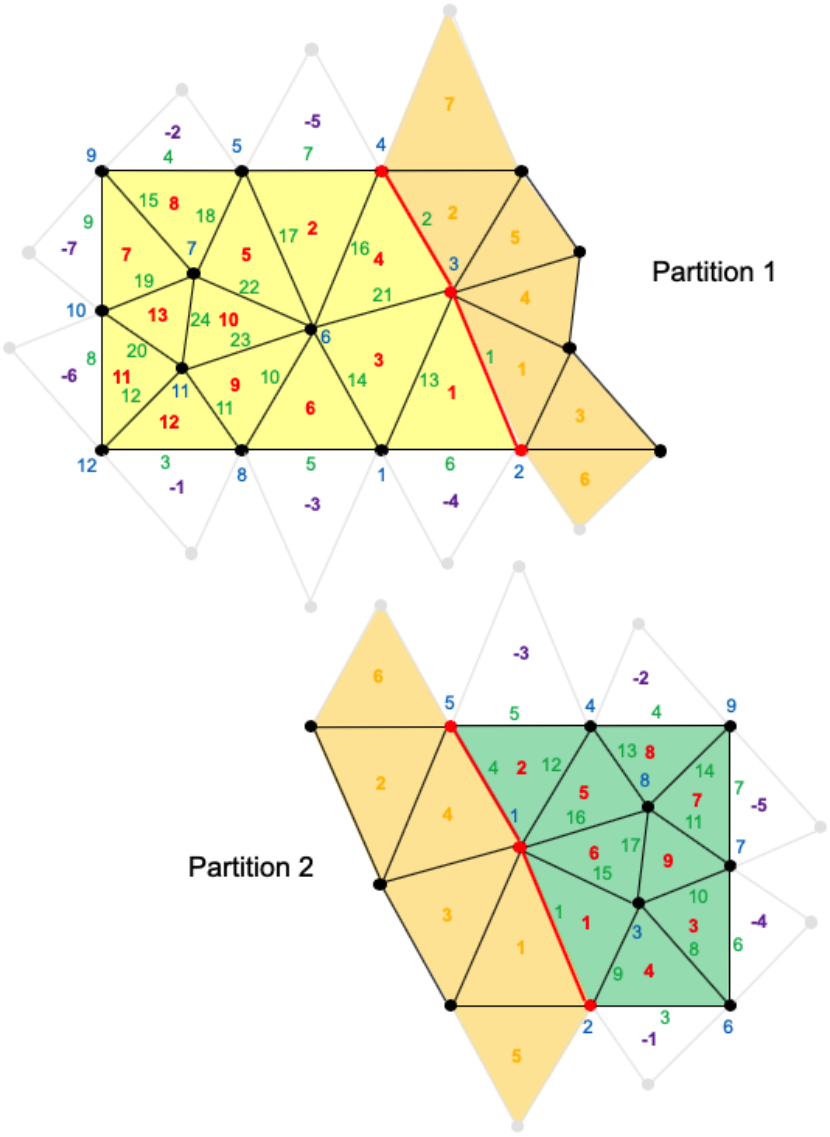

Figure 1.6: Demonstration of the treatment of periodic BC.

If we examine on the interior cells in Fig. 1.5b, each interior cell belongs to a unique partition (separated by the red line). Also, with the matching cell known for each periodic boundary face, we can know from where the periodic container cells should receive. The exact same data structure `itf` as in partitioning communication is used to represent the communication for periodic BC. `pinterf` (of the type `itf`) can be constructed in both partitions in the following way:

```
pinterf of Partition 1:
   2 receiving batches:
   receive-from id ---> 1 (itself) and 2
   No. of container cells ---> 4 (from id = 1) and 3 (from id = 2)
   periodic boundary faces to bear the container cells:
      face 4, 7, 3, 5 for from id = 1
```

```
      face 8, 9, 6 for from id = 2

   2 sending batches:
   sending-to id ---> 1 (itself) and 2
   No. of to-be-sent cells ---> 4 (to id = 1) and 3 (to id = 2)
   To-be-sent iterior cell indices:
      cell 8, 2, 6, 11 for to id = 1
      cell 7, 11, 1 for to id = 2
end pinterf

pinterf of Partition 2:
   2 receiving batches:
   receive-from id ---> 1 and 2 (itself)
   No. of container cells ---> 3 (from id = 1) and 2 (from id = 2)
   periodic boundary faces to bear the container cells:
      face 4, 5, 3 for from id = 1
      face 6, 7 for from id = 2

   2 sending batches:
   sending-to-id ---> 1 and 2 (itself)
   No. of to-be-sent cells ---> 3 (to id = 1) and 2 (to id = 2)
   To-be-sent iterior cell indices:
      cell 2, 7, 3 for to id = 1
      cell 8, 4 for to id = 2
end pinterf
```

In constructing the above data structure, we first identify the container cells in both partitions. These are "ghost cells" to be allocated and are not interior cells in the domain. Then, the sending cells (interior cells) are determined based on the container cells in each partition. It is noted that for periodic boundary cells, we do not need the cell-node connectivity as in partitioning cells. We only need the cell-face connectivity. We associate periodic boundary faces with their corresponding container cells. Then, the cell-node connectivity of the container cells is equivalent to the face-node connectivity of the periodic boundary faces. This can be applied to other type of boundary cells. The cell-node connectivity of boundary cells will always be equivalent to the face-node connectivity of the corresponding boundary faces.

As a final note, the communication of periodic BC must precede the communication of the partitioning. This is because we periodic BC communication will first fill in all the boundary cells, some of which can then be sent out via the partitioning communication.

1.5 Practical Guide

We shall give some practical instructions in the section regarding the generation and use of mesh files. The action items are summarized by the flow chart shown in Fig. 1.7.

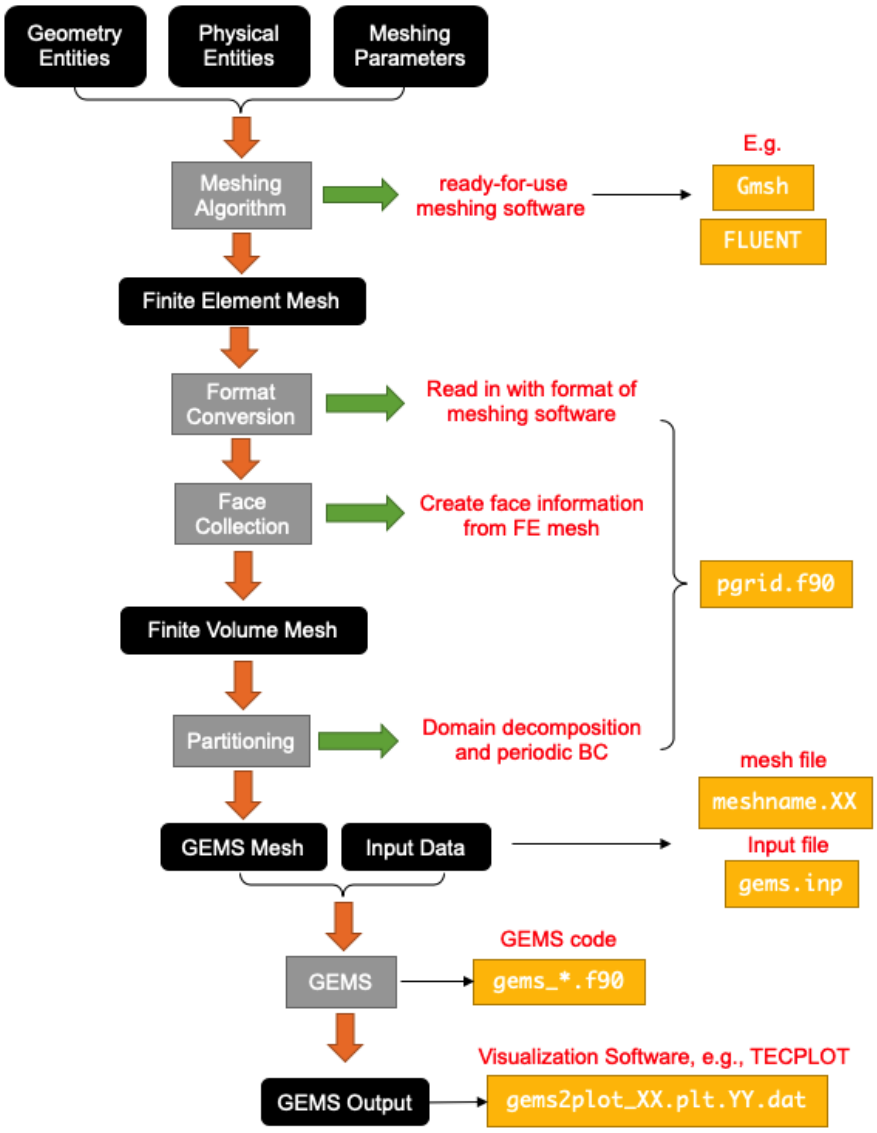

Figure 1.7: Flow chart of generating and using mesh files.

The first step is to generate a finite element mesh as described in section 1.1. We note that the mesh file is defined by three components: geometry entities, physical entities, and meshing parameters. The geometry entities refer to the point, lines, curves, etc. that define the geometry of the calculation domain. This is analogous to the CAD drawing you use for a design. Second, you associate the geometry entities with physical entities. For example, the geometry entities in Fig. 1.1(a) is 4 points, 4 lines and a rectangle. The lines that form the rectangle are associated with physical entities BC batch 1 (e.g., periodic boundaries), BC batch 2 (e.g., solid wall) and BC batch 3 (e.g., another solid wall). Finally, the meshing parameters that specify the way to "seed" the domain with nodes are defined. That can be how many nodes should be seeded along the lines, etc. The geometry entities, physical entities, and meshing parameters are then fed into the meshing algorithm to generate a finite element mesh like that in Fig. 1.1(b).

The meshing algorithm itself is a totally different science. We simply rely on ready-for-use toolboxes to generate the finite element mesh, rather than implement the mesh algorithm ourselves. `Gmsh` seems to be a good option for doing this, as it gives flexible automation with Gmsh scripts. However, you still need to learn how to draw points and lines, associated them with physical entities, as well as control the meshing schemes with meshing parameters. For `Gmsh`, those are specified in the `.geo` files, which are inputted into `Gmsh` to obtain the mesh (`.msh`) files.

Once we have the FE mesh file, we feed it into the code `pgrid.f90` which is a part of GEMS code and functions as a pre-processor to convert a FE mesh to the mesh file GEMS recognizes (referred to as the GEMS mesh). In `pgrid.f90`, the first task is to read in the FE mesh generated by other software (e.g. `Gmsh`). Unfortunately, the software usually writes a mesh file in complicated ways, and different software writes the mesh file differently. `pgrid.f90` has different interfaces for reading FE mesh files with different format, but mostly of them are obsolete. For example, the `GAMBIT` format was used in the past which is generated by `FLUENT`. But since `FLUENT` was acquired by `ANSYS`, we have lost the way to generate `GAMBIT` files. As of today, two formats are actively used: (1) Cartesian mesh format, which is generated without using any software, and (2) SU2 format, which is generated by `Gmsh` (choose `SU2` format when exporting).

The Cartesian mesh format is a simple format and it can be written in the following way (in FORTRAN, and use a 2D mesh as example):

```
do j=1, Nj
   do i=1, Ni
      write xn(i)
   end do
end do

do j=1, Nj
   do i=1, Ni
      write yn(j)
   end do
end do
```

Here `Ni` and `Nj` are the number of **nodes** (not cells) along each dimension. `xn(:)` and `yn(:)` are 1D arrays that contains the nodal coordinates along X and Y dimension, respectively. Besides this, the boundary condition information must be written in a separate file in the following way:

```
write ndim*2
write 1,Ni,1,1,0,0,label1
write 1,Ni,Nj,Nj,0,0,label2
write 1,1,1,Nj,0,0,label3
write Ni,Ni,1,Nj,0,0,label4
```

where `ndim` is the number of dimensions, and `label1` - `label4` are the labels for BC's on the four edges of the rectangle. Note that the edge is specified by the starting and ending

indices of the 1D arrays, and the zeros indicates that the third dimension is turned off in the 2D mesh. GEMS will then convert the Cartesian mesh into an essentially an unstructured mesh (in the code `pgrid.f90` by creating nodal coordinates and cell-node connectivity, etc.). This is because GEMS treats all the meshes as unstructured mesh to achieve its generality.

After the FE mesh is read in, the data type `face` will be used to generate the finite volume mesh, as described in section 1.2. After that, the mesh is partitioned to enable parallel computing. We note that this partitioning action includes both domain decomposition (described in section 1.3) and creating the ghost cells for the periodic BC's if there is any (described in section 1.4). These are all done in `pgrid.f90` and separate files will be generated for each partition. The GEMS mesh files are named as `meshname.XX` where `XX` stands for the indices of the partition (starting from 1, and then 2, 3, etc.).

A list of parameters that needs to be given in `pgrid.f90` are given in Table 1.1.

Table 1.1: List of key input parameters in pgrid.f90.

variable name	**meaning**
`gridfile`	the address of FE mesh files
`iftm`	the format of FE mesh file (structured, SU2, etc.)
`ndim`	dimension of the mesh
`nparts`	number of desired partitions
`im`	which partitioning algorithm
`npartsX, npartsY, npartsZ`	number of partitions along each dimension, if `im=7` is selected
`boundfile`	address of boundary condition file, if `iftm=1` is selected
`id`	specify which type of periodic BC is desired (0 means no periodic BC)
`npbc`	number of periodic BC labels (later will merge into one label for all periodic BC)
`lab_period,iax`	pair of periodic BC label and along which direction the periodicity is
`ngeom`	which mesh geometry to specify in TECPLOT file (1=triangle, etc.)

Couple of notes, (1) the ideal partitioning algorithm should the `METIS` algorithm which distributes the number of cells evenly among partitions. However, the FORTRAN interface of `METIS` has been lost and a poor man's partitioning algorithm `im = 7` is commonly used now. The poor man's partitioning algorithm divides the domain evenly according to the length along the X, Y and Z dimensions. Uneven distribution of cells will be caused if, for example, a non-uniform mesh size is applied. (2) When specifying multiple periodic BC's, a loop will be used to scan all the periodic BC's labels. Each label corresponds to a type of periodicity (cylindrical, 2D planar and 3D planar), and with each type of periodicity, the direction along which the periodicity occurs must also be specified. (3) a tecplot file will be generated for the GEMS mesh after the main programs of `pgrid.f90` are finished. In tecplot, the mesh geometry needs to be explicitly specified (by `ngeom`), whether it be triangle,

quadrilateral, tetrahedron, or brick. If the mesh is of other geometries, specify `ngeom` as 5 which basically treats all 2D meshes as quadrilaterals and all 3D meshes as bricks. This sometimes causes messy visualizations but this is the best that tecplot can do.

Now let's move on to the tasks after the GEMS mesh is generated (Fig. 1.7). The mesh files and the input data are fed into the GEMS code, after which the GEMS ouptut, i.e., the results of the CFD simulation can be obtained. The input data is specified in the file `gems.inp`. This is a manually written file in which a group of FORTRAN namelists are specified. These namelists record the material properties, numerical parameters, etc. used for the GEMS computation. The GEMS code consists of about 20 `f90` files with the suffix `gems_`. These files are different modules of GEMS and form the main programs of the CFD simulation. The output files are TECPLOT files with the name `gems2plot_XX.plt.YY.dat` where `XX` stands for the time strand and `YY` stands for the id of the partition. Note that each partition outputs independently, and therefore, if there are 4 partitions, 4 output files will be written (`YY` from 0 to 3) for one time strand `XX`. The data become difficult to manage when a large number of partition is used (e.g., 80). To deal with this, a python script is used to concatenate the 80 files for each time strand. Also, the tecplot script `preplot` is used convert all the `.dat` ASCII files into binary files to reduce the file size as well as the time it takes to load data into tecplot.

Chapter 2: Fundamentals of Fluid Mechanics

For a topic having such a variety of different facets like fluid mechanics, even the fundamentals are different in different contexts. Here we will introduce the fundamentals from the "computational" perspective. The essence is a set of partial differential equations that we are going to numerically solving. We shall properly define these PDE's and prepare them in a format ready for numerical computation.

2.1 Viewpoints of Describing Fluid Mechanics

The PDE's we are going to solve are the so-called conservation laws. That is, some **conservative variable** such as mass, momentum and energy should be conserved. Let's take the mass conservation for example. Here we focus our attention on a **material region**. That is a selection of particles (atoms, molecules) of material, and we track the same particles over time. The conservation of mass states that the mass of the tracked material region shall not be changed over time, as shown in 2.1.

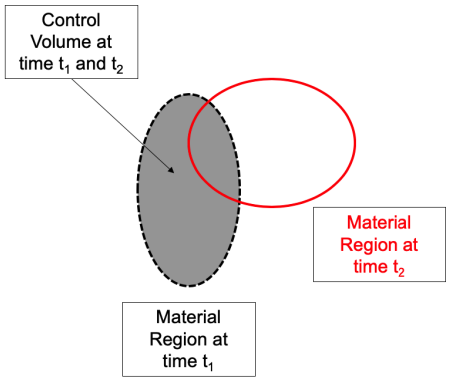

Figure 2.1: Material region and control volume. Black-dashed and red-circled region are the two material region at two time moments. The gray area overlaps with the material region at t_1 and represents the control volume which is fixed regardless of time.

Let's express the mass conservation in the format of **integral equation**:

$$\frac{\partial}{\partial t}\int_{\Omega_M(t)} \rho d\Omega = 0 \tag{2.1}$$

Here Ω_M stands the volume of the material region which can be changing with time t. ρ is

the density and a function of time and space, and $d\Omega$ is a infinitesimal volume in 3D. Next, we want to re-write Eqn. 2.1 in the context of **control volume**, instead of the material region. A control volume is a fixed volume of space and do not change with time. Mass can flow into or out of the control volume. Typically, we only care a specific type of control volume. That is the cells we introduced in chapter 1. We will be dealing with tons of control volumes as each cell represents a fixed volume of space and is viewed as one control volume. The difference of control volume and material region should be emphasized. The material region is the volume of space that encompasses a certain set of particles being tracked, so the material region changes with time depending on the velocities of the particles. The control volume is a fixed volume of space, and particles can freely flow into or out of the control volume. Technically, the control volume can also be set to be moving, but let's just set it to be stationary all the time. These two concepts represent two perspectives of describing the mechanics of particles. The material region is called the **Lagrangian point of view** and the control volume is referred to as the **Eulerian point of view**.

How to bridge the two point of views of describing mechanics? Let's find a time moment where the material region and control volume are overlaid on one another, as shown in Fig. 2.1. If we take the Eulerian perspective, the mass within the **control volume** should be changing and the rate should be equal to the net mass rate flowing into/out of the control volume. Therefore, writing in the format of integral equation, we have:

$$\int_{\Omega} \frac{\partial \rho}{\partial t} d\Omega + \int_{\partial\Omega} \rho (V \cdot n) d\sigma = 0 \tag{2.2}$$

Here Ω is the control volume (we dropped the subscript M to differentiate control volume and material region). Note that we can move the partial derivative inside the integral since Ω does not change with time. ρ is still density as a function of time and space. V is the velocity (a 3D vector) and n is the normal unit vector (also a 3D vector) on the 2D boundary of the 3D control volume (written as $\partial\Omega$. The dot product $V \cdot n$ represents the normal component of velocity on the boundary $\partial\Omega$. Finally, $d\sigma$ stands for the infinitesimal surface area on $\partial\Omega$. One note regarding notation, we do not use the arrow or bold font to differentiate vector, tensor and scalar. This clarity of the notation is sacrificed to trade for conciseness in our equations.

The two equations Eqn. 2.1 and 2.2 are equivalent to each other. They describe the same conservation law of mass but with two point of views. Eqn. 2.1 is the Lagrangian view and Eqn. 2.2 is the Eulerian view. We will only use the Eulerian point of view of describe fluid mechanics. The equivalence of the two point of views can be proved mathematically based on the Leibnitz's theorem [1]. The Leibnitz's theorem states that for any quantity Q:

$$\frac{\partial}{\partial t} \int_{\Omega_M(t)} Q d\Omega = \int_{\Omega} \frac{\partial Q}{\partial t} d\Omega + \int_{\partial\Omega} Q (V \cdot n) d\sigma \tag{2.3}$$

In the context of fluid mechanics, Q can be mass, momentum and energy.

2.2 Governing equations

For now on, we will be focusing on using control volume to describe the conservation equations (Eulerian viewpoint). We have already introduced the mass conservation equation as Eqn. 2.2. Next, momentum conservation. In 3D, the momentum along the three dimensions should all be "conserved" which gives 3 momentum conservation equations. Unlike the mass conservation, the momentum of the control volume can be changed by forces. This is Newton's second law. We write the integral form of momentum conservation equation as:

$$\int_\Omega \frac{\partial(\rho V_i)}{\partial t} d\Omega + \int_{\partial\Omega} (\rho V_i)(V \cdot n) d\sigma = \int_\Omega \rho f_i d\Omega + \int_{\partial\Omega} R_i d\sigma \tag{2.4}$$

On the left-hand-side (LHS), the first term is the temporal rate of the momentum along the i dimension ($i = x, y, z$), integrated over the control volume. The second term is the momentum flowing into or out of the control volume. Note that the entire left-hand-side can be interpreted in the Lagrangian viewpoint as the temporal rate of momentum of the **material region**. Use Leibnitz's theorem, Eqn. 2.3. The right-hand-side stands for the force. The first term on the right-hand-side (RHS) is the **body force** (the component along the i direction), while the second term is the **surface force**. We note that the body force has the unit of N/kg or the unit of acceleration m^2/s. The surface force has the unit of N/m^2. The most common body force is the gravitational force $g = [0, -9.81, 0]^T$

The surface force needs our special attention. To describe a surface force, we need to use **stress tensor**. Surface force can be alternatively referred to as the surface stress. Stress tensor is a matrix T, and the surface force is given by $R = Tn$. This is a matrix multiplication where n and R are column vectors. We will assume throughout this document a vector is a column vector, unless otherwise specified. This can also be written as $R = (n \cdot T)^T$. I will avoid using matrix multiplications and instead use dot products (inner product) or outer product in this document. Note the dot (inner) product of matrices A and B are defined as $A \cdot B = A^T B$. The multiplication of two matrices can be expressed as $AB = A^T \cdot B$. The outer product of two matrices are defined as: $A \otimes B = AB^T$.

The stress tensor has two components, pressure and viscous stress, expressed as:

$$T = -pI + \tau$$

where p is the **thermodynamic pressure**, I is a 3 by 3 unit matrix (in 3D), and τ is the viscous stress tensor which is a matrix as a function of the gradient of velocities. This relationship is referred to as **constitutive relations** which will be discussed later.

Next, let's consider the conservation of energy. Again, writing as integral equation:

$$\int_\Omega \frac{\partial(\rho e_{tot})}{\partial t} d\Omega + \int_{\partial\Omega} (\rho e_{tot})(V \cdot n) d\sigma = \int_\Omega \rho f \cdot V d\Omega + \int_{\partial\Omega} (n \cdot T)^T \cdot V d\sigma - \int_{\partial\Omega} (n \cdot q) d\sigma \tag{2.5}$$

Here e_{tot} is the **total energy**, $e_{tot} = e + \frac{1}{2}V^2$, where e is the **specific** internal energy, or energy per unit mass. The two terms combined on the LHS stands for the temporal rate of total energy of the mass region, again, Leibnitz's theorem (Eqn .2.3). The first two terms

on the RHS stand for the **work** done by the body force and surface force per unit time, or the **power**. The last term of the RHS stands for the heat transfer, or heat flux flowing into or out of the control volume, where q is the heat flux vector (with the unit of W/m^2. The heat flux is a function of temperature gradient. Such a relationship is also part of the constitutive relations. The long equation (Eqn. 2.5) is nothing other than the **first law of thermodynamics**, written as "$dU = \delta Q - \delta W$" in typical thermodynamic textbooks.

Eqn. 2.2, 2.4, and 2.5 constitutes the **governing equations** of fluid mechanics, written in the form of integral equations. This set of equations is the basic set while more governing equations need to be included when considering additional physics, e.g., conservation of chemical species. That being said, we limit our discussion to this set of basic equations in this chapter.

2.3 Constitutive Relations

There are essentially five equations from Eqn. 2.2, 2.4, and 2.5. Therefore, we need to pick five independent variables, referred to as the **primitive variables**, and express all the variables in Eqn. 2.2, 2.4, and 2.5 as functions of the independent. Such relations are generally referred to as the **constitutive relations**. We will be exclusively using (p, V, T) as the primitive variables in this document. Note that the choice of primitive variables can be arbitrary, but different choice may lead to entirely different constitutive relations. The choice of (p, V, T) is because that it is a convention to express other properties of a material as functions of p, V, T in, for example, material handbooks and databases.

The **thermodynamic relations** include $e = e(p, T)$ (internal energy), $\rho = \rho(p, T)$, and sometimes the **enthalpy**, $h = e + p/\rho$. The **total enthalpy** can be expressed as $h_{tot} = h + \frac{1}{2}V^2$. These relations are for the "state functions" and are often referred to as the **equation of state** (EOS).

The **kinetic relations** describe the "transfer" of some quantity. For example, the transfer of heat is described by the **Fourier's law**:

$$q = -k\nabla T$$

where k is the thermal conductivity. Note that k is referred to as (one of) the **transport properties** and can be a function of primitive variables, $k = k(p, T)$. Also, Fourier's law only governs the **conduction** of heat transfer. Convection and radiation can be introduced by including additional kinetic relations. But this is beyond the scope of the current discussion. The transfer of momentum is described by the **Newton's viscosity law**:

$$\tau = \left(-\frac{2}{3}\mu\nabla \cdot V\right) I + 2\mu S$$

where μ is the viscosity and can be a function of primitive variables, $\mu = \mu(p, T)$. $\nabla \cdot V$ is the divergence of velocity. Note that the nabla operator can be also treated as a column vector: $\nabla = [\partial/\partial x, \partial/\partial y, \partial/\partial z]^T$. Again, I is a 3 by 3 unit matrix, and S is the **strain rate tensor**:

$$S = \frac{1}{2}\begin{bmatrix} 2\frac{\partial u}{\partial x} & \frac{\partial v}{\partial x}+\frac{\partial u}{\partial y} & \frac{\partial w}{\partial x}+\frac{\partial u}{\partial z} \\ \frac{\partial u}{\partial y}+\frac{\partial v}{\partial x} & 2\frac{\partial v}{\partial y} & \frac{\partial w}{\partial y}+\frac{\partial v}{\partial z} \\ \frac{\partial u}{\partial z}+\frac{\partial x}{\partial w} & \frac{\partial w}{\partial y}+\frac{\partial v}{\partial z} & 2\frac{\partial w}{\partial z} \end{bmatrix}$$

where (u, v, w) are the three components of the velocity vector V. One final note regarding the thermodynamic and kinetic relations. The term "thermodynamic" and "kinetic" are from material science. Basically, "thermodynamic" is associated with the states of equilibrium while "kinetic" is associated with the states in between equilibrium, and therefore, with the gradient of thermodynamic properties. This is a hand-wavy distinction of these two categories of constitutive relations, based on concepts in material science.

Up to now, we can express all the quantities in Eqn. 2.2, 2.4, and 2.5 as functions of the primitive variables (p, V, T). The body force f may be the exception, but for now we can simply view it as a constant as for the gravitational force.

2.4 Differential Governing Equations

In this section we brief discuss the differential form of governing equations. Although this form is not our focus in numerical implementations, it can be convenient to use this form in analysis. To derive the differential form, we need to use Gauss's theorem

$$\int_\Omega \nabla\phi \, d\Omega = \int_{\partial\Omega} n\phi \, d\sigma, \quad \phi \text{ is scalar} \tag{2.6a}$$

$$\int_\Omega \nabla \cdot v \, d\Omega = \int_{\partial\Omega} n \cdot v \, d\sigma, \quad v \text{ is vector} \tag{2.6b}$$

$$\int_\Omega \nabla \cdot T \, d\Omega = \int_{\partial\Omega} n \cdot T \, d\sigma, \quad T \text{ is matrix (tensor)} \tag{2.6c}$$

Again, the nabla operator is treated as a column vector $\nabla = [\partial/\partial x, \partial/\partial y, \partial/\partial z]^T$, and note the LHS and RHS of Eqn. 2.6c are both row vectors. Eqn. 2.6c can be transposed to give the column vector form.

Now, we can use Eqn. 2.6 to transform the surface integrals to volume integrals in Eqn. 2.2, 2.4, and 2.5. Then, we can drop the integral and keep the intergrand to obtain the differential equations. For example, the mass conservation can be written as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho V) = 0 \tag{2.7}$$

by applying Eqn. 2.6b and let $v = \rho V$. To derive momentum conservation equation, we need to group the terms inside the surface integral to form tensors. First, we write Eqn. 2.4 as:

$$\int_\Omega \frac{\partial(\rho V)}{\partial t} d\Omega + \int_{\partial\Omega} \left(n \cdot (\rho V \otimes V)\right)^T d\sigma = \int_\Omega \rho f d\Omega + \int_{\partial\Omega} (n \cdot T)^T d\sigma \tag{2.8}$$

where $T = -pI + \tau$, and note that $\rho V \otimes V$ is a matrix. Then, apply Eqn. 2.6c to the surface integrals to get:

$$\frac{\partial(\rho V)}{\partial t} + \left(\nabla \cdot (\rho V \otimes V)\right)^T = \rho f + (-\nabla p) + (\nabla \cdot \tau)^T \tag{2.9}$$

Note that we applied the transpose operation to keep very vector as a column vector. Finally, to derive the energy conservation equation, we observe that the work done by the surface force can be rewritten as $(n \cdot T)^T \cdot V = n \cdot (T^T \cdot V)$. That is, the dot product of n and some matrix. Therefore, we can apply Eqn. 2.6c and obtain:

$$\frac{\partial(\rho e_{tot})}{\partial t} + \nabla \cdot (\rho e_{tot} V) = \rho f \cdot V + (-\nabla \cdot (pV)) + \nabla \cdot (\tau^T \cdot V) - \nabla \cdot q \tag{2.10}$$

Now we have the set of differential equations, Eqn. 2.7, 2.9, and 2.10, that governs the conservation of mass, momentum and energy.

2.5 Matrix Representation

In this section, we will prepare the set of governing equations in integral form, Eqn. 2.2, 2.8, and 2.5 in a format that is more suitable for numerical computation. Let's first seperate the pressure and the viscous stress from the surface stress tensor T, and rewrite the equations as follows:

$$\int_\Omega \frac{\partial \rho}{\partial t} d\Omega + \int_{\partial\Omega} n \cdot (\rho V) d\sigma = 0 \tag{2.11a}$$

$$\int_\Omega \frac{\partial(\rho V)}{\partial t} d\Omega + \int_{\partial\Omega} (n \cdot (\rho V \otimes V))^T d\sigma = \int_{\partial\Omega} (n \cdot (-pI))^T d\sigma + \int_{\partial\Omega} (n \cdot \tau)^T + \int_\Omega \rho f d\Omega \tag{2.11b}$$

$$\int_\Omega \frac{\partial(\rho e_{tot})}{\partial t} d\Omega + \int_{\partial\Omega} n \cdot (\rho e_{tot} V) d\sigma = \int_{\partial\Omega} n \cdot (-pV) d\sigma + \int_{\partial\Omega} n \cdot (\tau^T \cdot V) d\sigma - \int_{\partial\Omega} (n \cdot q) d\sigma + \int_\Omega \rho f \cdot V d\Omega \tag{2.11c}$$

Notice that for the energy equation Eqn. 2.5, we used:

$$(n \cdot T)^T \cdot V = n \cdot (T^T \cdot V) = n \cdot ((-pI + \tau^T) \cdot V) = n \cdot (-pV) + n \cdot (\tau^T \cdot V)$$

We observe a similarity in Eqns 2.11 that all the surface integrals has the form of the norm vector n, dot product with some vector or tensor. Also, we want to distinguish the terms related to gradients (e.g., τ and q) from those which does not. For that, we can move the pressure-related terms from the RHS to the LHS to give:

$$\int_\Omega \frac{\partial \rho}{\partial t} d\Omega + \int_{\partial\Omega} n \cdot (\rho V) d\sigma = 0 \tag{2.12a}$$

$$\int_\Omega \frac{\partial(\rho V)}{\partial t} d\Omega + \int_{\partial\Omega} (n \cdot (\rho V \otimes V + pI))^T d\sigma = \int_{\partial\Omega} (n \cdot \tau)^T + \int_\Omega \rho f d\Omega \tag{2.12b}$$

$$\int_\Omega \frac{\partial(\rho e_{tot})}{\partial t} d\Omega + \int_{\partial\Omega} n \cdot (\rho h_{tot} V) d\sigma = \int_{\partial\Omega} n \cdot (\tau^T \cdot V) d\sigma - \int_{\partial\Omega} (n \cdot q) d\sigma + \int_\Omega \rho f \cdot V d\Omega \tag{2.12c}$$

Note that for Eqn. 2.12c, we used $\rho e_{tot} + p = \rho h_{tot}$. Now, with the form of Eqn. 2.12, we can group terms together to give the **matrix representation** of the governing equations:

$$\int_\Omega \frac{\partial Q_c}{\partial t} d\Omega + \int_{\partial\Omega} (n \cdot F_c)^T d\sigma = \int_{\partial\Omega} (n \cdot F_v)^T d\sigma + \int_\Omega S d\Omega \tag{2.13}$$

where the terms in Eqn. 2.13 are as follows:

$$Q_c = \begin{bmatrix} \rho \\ \rho V \\ \rho e_{tot} \end{bmatrix}_{5\times 1} \qquad F_c = \begin{bmatrix} \rho V & \rho V \otimes V + pI & \rho h_{tot} V \end{bmatrix}_{3\times 5}$$

$$F_v = \begin{bmatrix} 0 & \tau & \tau^T \cdot V - q \end{bmatrix}_{3\times 5} \qquad S = \begin{bmatrix} 0 \\ \rho f \\ \rho f \cdot V \end{bmatrix}_{5\times 1}$$

Again, all the vectors are expressed as column vectors. Eqn. 2.13 can also be succinctly written as:

$$\int_\Omega \frac{\partial Q_c}{\partial t} d\Omega + \int_{\partial\Omega} (n \cdot (F_c - F_v))^T d\sigma = \int_\Omega S d\Omega \tag{2.14}$$

The corresponding differential form of the matrix representation can be derived by simply applying Eqn. 2.6c:

$$\frac{\partial Q_c}{\partial t} + (\nabla \cdot (F_c - F_v))^T = S \tag{2.15}$$

Eqn. 2.15 is the one used in [2], but the integral form will be the foundation for numerical computation.

Now let's briefly discuss the importance of such matrix representation Eqn. 2.14. Here, Q_c is referred to as the **conservative variables**, F_c the **convective flux**, F_v the **viscous flux** and S the source term. It is a succinct representation of a general form of conservation law. It states that the temporal rate of a vector variable Qc is due to the fluxes due to translation of such variable (F_c), the gradient of such variable (F_v) as well as the external source addition/destruction of such variable (S). With this form, we can conveniently add more conservative variables to the vector Q_c and modify accordingly the convective flux, viscous flux and the source term.

The control volume Ω in Eqn. 2.14 will be millions of cells in the mesh. For each cell, we write down Eqn. 2.14, with 5 unknowns in Q_c. Then, we will have millions of unknowns and millions of equations for the entire calculation domain. Notice that the boundary of the cell $\partial\Omega$ will be composed of faces. This foreshadows the surface integrals on the boundary of a cell will be approximated by summations over the faces of the cell.

It is more convenient to deal with the primitive variables $Q_p = [p, V, T]^T$, rather than the conservative variables Q_c. Therefore, we can express Eqn. 2.14 as:

$$\int_\Omega \Gamma \frac{\partial Q_p}{\partial t} d\Omega + \int_{\partial\Omega} (n \cdot (F_c(Q_p) - F_v(Q_p)))^T d\sigma = \int_\Omega S(Q_p) d\Omega, \text{ where } \Gamma = \frac{\partial Q_c}{\partial Q_p} \tag{2.16}$$

That is, expressing every term as functions of the primitive variable Q_p. Here, Γ is referred to as the **conservative jacobian** which connects primitive variable and conservative variable.

2.6 Preconditioning: Motivation

In this section, we introduce some analysis on the system of governing equations Eqn. 2.14 or 2.15 and reveal its properties. Importantly, the governing equations can be viewed as a superimposition of wave equations, each with its propagation speed. We shall see that the propagation speed of different waves can have drastically different magnitude under certain circumstances (subsonic flow). In this scenario, the different wave speeds give difficulties in numerically solving the governing equations. This problem is referred to as the **ill-conditioning** of the system, and we will introduce the **preconditioning** method to such difficulties.

First, let's reveal the wave-nature of the governing equations by considering the differential form of the **one-dimensional Euler equation** which is often used for analysis. Starting from Eqn. 2.15, we make the equations 1D, and remove the viscous flux (inviscid flow) and the source term (neglected gravity), and obtain the following equation:

$$\frac{\partial Q_c}{\partial t} + \frac{\partial E}{\partial x} = 0; \quad Q_c = \begin{bmatrix} \rho \\ \rho u \\ \rho h_{tot} - p \end{bmatrix}; \quad E = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho h_{tot} u \end{bmatrix} \tag{2.17}$$

Here, E is the 1D convective flux and u is the 1D velocity. Notice that we used $\rho h_{tot} - p = \rho e_{tot}$ to replace any internal energy related terms with enthalpy related terms. This is simply a convention that enthalpy is more popular than internal energy in material properties handbooks. It should be pointed out that although the 1D Euler equation Eqn. 2.17 is significantly simplified from the original differential governing equation Eqn. 2.15, the essence of the governing equation is retained. It will be straightforward to show that the 1D Euler equation is a superimposition of wave equations.

Now let's use primitive variable $Q_p = [p, u, T]^T$ to express the 1D Euler equation as:

$$\Gamma \frac{\partial Q_p}{\partial t} + A_c \frac{\partial Q_p}{\partial x} = 0; \quad \Gamma = \frac{\partial Q_c}{\partial Q_p}, \; A_c = \frac{\partial E}{\partial Q_p} \tag{2.18}$$

where A_c is the **convective flux jacobian** and Γ is the conservative jacobian. These jacobians can be explicitly evaluated by taking the partial derivatives of material properties with respect to the primitive variables, as follows:

$$\Gamma = \begin{bmatrix} \rho_p & 0 & \rho_T \\ \rho_p u & \rho & \rho_T u \\ \rho_p h_{tot} + \rho h_p - 1 & \rho u & \rho_T h_{tot} + \rho h_T \end{bmatrix}$$

$$A_c = \begin{bmatrix} \rho_p u & \rho & \rho_T u \\ \rho_p u^2 + 1 & 2\rho u & \rho_T u^2 \\ (\rho_p h_{tot} + \rho h_p) u & \rho h_{tot} + \rho u^2 & (\rho_T h_{tot} + \rho h_T) u \end{bmatrix}$$

Here we used subscript notation a_b to represent the partial derivative of b with respect to a. We point out that there are in total 4 partial derivatives in Γ and A_c which are ρ_p, ρ_T, h_p and h_T. They all have significant physical meanings and are fairly accessible in material property handbooks.

We can further simply Eqn. 2.18 by multiplying the LHS with Γ^{-1} and group $\Gamma^{-1}A_c$ to be a matrix A:

$$\frac{\partial Q_p}{\partial t} + A\frac{\partial Q_p}{\partial x} = 0; \quad A = \Gamma^{-1}A_c \tag{2.19}$$

Eqn. 2.19 already has the "form" of a wave equation. Notice that the three components of Q_p are intertwined through the matrix A, typically referred to as the **system matrix** of a wave equation. This coupling can be removed by performing the diagonalization of the system matrix A:

$$A = M\Lambda M^{-1}; \quad \Lambda = \text{diag}\{\lambda_1, \lambda_2, \lambda_3\}$$

Here Λ is a diagonal matrix storing the eigenvalues of A and M stores the eigenvectors in its column vectors. Note that both Λ and M are analytical expression that can be explicitly evaluated. Next, we manufacture a variable $\widehat{Q}$, such that:

$$M = \frac{\partial Q_p}{\partial \widehat{Q}}$$

This is to transform Eqn. 2.19 to:

$$M\frac{\partial \widehat{Q}}{\partial t} + AM\frac{\partial \widehat{Q}}{\partial x} = 0$$

Then, multiply the LHS by M^{-1} and realize $\Lambda = M^{-1}AM$ to obtain:

$$\frac{\partial \widehat{Q}}{\partial t} + \Lambda\frac{\partial \widehat{Q}}{\partial x} = 0 \tag{2.20}$$

We refer $\widehat{Q}$ as the **characteristic variable**. Note that $\widehat{Q}$ is purely manufactured and does not necessarily have any physical meaning. It is defined by a differential relation with Q_p and in most of the cases, does not even have a closed-form expression. However, by transforming Q_p to $\widehat{Q}$, the wave nature of the governing equation can be better revealed as in Eqn. 2.20. We can also write it as:

$$\begin{cases} \frac{\partial \widehat{Q}_1}{\partial t} + \lambda_1\frac{\partial \widehat{Q}_1}{\partial x} = 0 \\ \frac{\partial \widehat{Q}_2}{\partial t} + \lambda_2\frac{\partial \widehat{Q}_2}{\partial x} = 0 \\ \frac{\partial \widehat{Q}_3}{\partial t} + \lambda_3\frac{\partial \widehat{Q}_3}{\partial x} = 0 \end{cases}$$

As we can see, each component of the characteristic variable, $\widehat{Q}_i$ $(i = 1, 2, 3)$ propagates like a wave with the propagation speed of λ_i. If λ_i were constant (or approximately as constant), these would have been 1D linear wave equations with analytical solution. In such a case, Eqn. 2.19 can be solved (or approximately solved) by transforming back from $\widehat{Q}$ to Q_p.

In reality, λ_i's are not constant. They can be expressed as some functions of the primitive variable. These functions can be derived by diagonalizing the system matrix $\Gamma^{-1}A_c$. Unfortunately, such a derivation will be extremely tedious as the system matrix is composed of long expressions of the primitive variable. Therefore, this task is usually done by symbolic operations of some computer program such as MATLAB. After doing so, the wave propagation speeds λ_i's are found out to be:

$$\lambda_1 = u \tag{2.21a}$$

$$\lambda_2 = u + \sqrt{\frac{\rho h_T}{\rho_T + \rho h_T \rho_p - \rho h_p \rho_T}} \tag{2.21b}$$

$$\lambda_3 = u - \sqrt{\frac{\rho h_T}{\rho_T + \rho h_T \rho_p - \rho h_p \rho_T}} \tag{2.21c}$$

The **acoustic speed**, or **sound speed** can be defined as:

$$c = \sqrt{\frac{\rho h_T}{\rho_T + \rho h_T \rho_p - \rho h_p \rho_T}} \tag{2.22}$$

such that the three wave speeds are:

$$\lambda_1 = u,\ \lambda_2 = u + c,\ \lambda_3 = u - c$$

We refer to the wave associated with λ_1 as the **particle wave** because its propagation speed is equal to the transnational velocity of particles of the material, u. We refer to the waves associated with λ_2 and λ_3 as the **acoustic waves** as they involves the intrinsic acoustic property of the material, the sound speed c.

It follows from Eqn. 2.21 that, when the particle speed u is comparable to the acoustic speed c, all the wave speeds are comparable. This is the case in transonic and supersonic flows typically studied in aerodynamics. However, if the particle speed u is small compared to c, as typical in subsonic flows, there can be a large difference between the wave speeds. Since wave speeds are the eigenvalues of the system matrix $\Gamma^{-1}A_c$, the term "ill-conditioned" is borrowed from linear algebra to describe this situation, where the magnitude of eigenvalues of a matrix is drastically different.

The ill-conditioning of the system brings a variety of difficulties when numerically solving the governing equation Eqn. 2.18. As we have not discussed any aspects regarding numerical computation, we only qualitatively describe the difficulties here. One of the difficulties is that the numerical errors tend to be magnified in an ill-conditioned system, which often causes the code to crash. To avoid the crash, more incremental steps must be taken in the solution process such that the errors in each step is small and can be contained. But this can significantly increases the computational cost, or in other words, "the code runs too slow". These difficulties are intrinsic due to the system eigenvalues having disparate magnitude.

The philosophy of preconditioning is to alter the eigenvalues of the system such that all the eigenvalues have comparable magnitude. To do that we modify the temporal term in the original governing equation Eqn. 2.18 to write:

$$\Gamma_p \frac{\partial Q_p}{\partial t} + A_c \frac{\partial Q_p}{\partial x} = 0$$

Note that the original conservative jacobian Γ is modified as Γ_p. By doing so, the new system matrix $\Gamma_p^{-1}A_c$ will have a new set of eigenvalues $\{\lambda_1, \lambda_2, \lambda_3\}$ different than those in Eqn. 2.21. The new eigenvalues will have comparable magnitude and make the system

properly conditioned.

One may be wondering whether the preconditioning has changed the governing equation, and therefore, gives unphysical solutions. Not for the **steady state solution** which does not change with time. Indeed, for a steady state problem, the governing equation in Eqn. 2.18 could be written as:

$$A_c \frac{\partial Q_p}{\partial x} = 0 \tag{2.23}$$

since $\partial Q_p/\partial t = 0$ for steady state solution. Therefore, altering the conservative jacobian in the temporal term will not have any effects on the steady state solution. For a steady state problem, the time evolution does not have to be physical, as long as the temporal rate eventually vanishes and Eqn. 2.23 is satisfied. In this sense, the time evolution in steady state problems can be understood as a **pseudo time** evolution. We often use τ instead of the t to emphasize this fact:

$$\Gamma_p \frac{\partial Q_p}{\partial \tau} + A_c \frac{\partial Q_p}{\partial x} = 0$$

and this strategy of solving steady state problems is referred to as the **time marching method**.

But what if the solution is unsteady as Q_p can evolve with time? The same philosophy can be elegantly applied. We can introduce a second set of time evolution which, in its steady state, approaches to the original unsteady governing equation. Such scheme is referred to as the **dual time scheme**. For example, we can write the governing equation as:

$$\Gamma_p \frac{\partial Q_p^{n+1}}{\partial \tau} + \Gamma \frac{Q_p^{n+1} - Q_p^n}{\Delta t} + A_c \frac{\partial Q_p^{n+1}}{\partial x} = 0$$

where Q_p^n is the solution at the n^{th} time moment t^n, which is assumed to be known. Q_p^{n+1} is the unknown to be solved at the next time moment t^{n+1}, and $\Delta t = t^{n+1} - t^n$ is the time step size. Note that Q_p^{n+1} is a function of x and τ but is independent of the physical time t, $Q_p^{n+1} = Q_p^{n+1}(x, \tau)$. In its steady state solution (of pseudo time τ), the temporal term vanishes, $\partial Q_p^{n+1}/\partial \tau = 0$, and the governing equation goes back to:

$$\Gamma \frac{Q_p^{n+1} - Q_p^n}{\Delta t} + A_c \frac{\partial Q_p^{n+1}}{\partial x} = 0$$

Once we find the solution Q_p^{n+1}, we can proceed to the next physical time step. In the next physical time step, again, we need to solve a steady state problem in the pseudo time τ to obtain a solution Q_p^{n+2}, etc. The detailed discussion regarding the dual time scheme will be given the next chapter.

Chapter 3: Algorithm

3.1 Spatial and Explicit Temporal Discretization

In this chapter, we will be using the data structure introduced in chapter 1 to solve the governing equations introduced in chapter 2. Specifically, the physical domain is discretized by the mesh, e.g., in Fig. 1.3. Each cell is a control volume in which we can write the governing equation, e.g., Eqn. 2.16 by applying the cell volume as Ω and the cell boundary (surrounding faces) as $\partial\Omega$. By doing so, the number of governing equations we can write down is equal to the number of cells in the mesh. Now, we need to **discretize** the governing equation in order to produce numerical solutions. Let's first rewrite the governing equations as (based on Eqn. 2.13, 2.14, and 2.16):

$$\begin{aligned} &\int_\Omega \Gamma \frac{\partial Q_p}{\partial t} d\Omega + \int_{\partial\Omega} [F_{cn}(Q_p, n) - F_{vn}(Q_p, n)] d\sigma = \int_\Omega S(Q_p) d\Omega \\ &\text{where } F_{cn} = \begin{bmatrix} \rho V_n \\ \rho V_n V + pn \\ \rho V_n h_{tot} \end{bmatrix}, \quad F_{vn} = \begin{bmatrix} 0 \\ \tau \cdot n \\ n \cdot (\tau^T \cdot V) - n \cdot q \end{bmatrix}, \quad V_n = V \cdot n \end{aligned} \tag{3.1}$$

Here we created the terms F_{cn} and F_{vn} for the convective and viscous flux along the normal direction n of the control volume surface. Note that we have two volume integrals and one surface integral. We can use the **volume-average** of primitive variable of each cell to approximate the volume integrals:

$$\int_{\Omega_I} \Gamma \frac{\partial Q_p}{\partial t} d\Omega \approx \Gamma(Q_{p,I}) \frac{\partial Q_{p,I}}{\partial t} \Omega_I \tag{3.2a}$$

$$\int_{\Omega_I} S(Q_p) d\Omega \approx S(Q_{p,I}) \Omega_I \tag{3.2b}$$

Here we use capital letter I to index the cells. Ω_I is the volume of the cell and $Q_{p,I}$ is the volume-average of the primitive variable for cell I. For the surface integrals, we want to use the volume-average of cell primitive variable to provide an approximation:

$$\int_{\partial\Omega_I} F_{cn}(Q_p, n) d\sigma \approx \sum_{i \in I_{sf}} \mathcal{F}(Q_{pL,i}, Q_{pR,i}, n_{Ii}) \sigma_i \tag{3.3a}$$

$$\int_{\partial\Omega_I} F_{vn}(Q_p, n) d\sigma \approx \sum_{i \in I_{sf}} F_{vn}(Q_{p,i}, (\nabla \otimes Q_p)_i, n_{Ii}) \sigma_i \tag{3.3b}$$

We use lower case letter i to index the faces. Here, I_{sf} is the set of faces surrounding cell I, $Q_{pL,i}$ is referred to as the **left state** of face i, $Q_{pR,i}$ is the **right state** of face i, $Q_{p,i}$ is the

face-average of primitive variable on the face i, $(\nabla \otimes Q_p)_i$ is the **face gradient** of primitive variable on the face i, n_{Ii} is the normal vector of face i pointing **out of** cell I, and finally, σ_i is the area of face i. Notice that there is a significant difference in the calculation of convective and viscous flux. For the convective flux, a **numerical flux function**, $\mathcal{F}$, is defined which is different from the analytical expressions of F_{cn} as in Eqn. 3.1. The computation of $\mathcal{F}$ will rely on the determination of the left and right state. For the viscous flux, the analytical expression F_{vn} as in Eqn. 3.1 is kept, and the face-based terms, $Q_{p,i}$ and $(\nabla \otimes Q_p)_i$ need to be determined before the viscous flux can be computed. We note that the approximation in Eqn. 3.3 utilizes the face-based terms with a subscript i while the approximation in Eqn. 3.2 utilizes the cell-based terms with a subscript I. We emphasize that the cell-based terms will be first calculated, and then, the face-based terms can be **interpolated** from the cell-based terms.

The numerical flux function deserves some extra comments. To compute a numerical (convective) flux, we first need to construct the left and right state with respect to a face. Next, the state variables are plugged into the flux function $\mathcal{F}$ to produce the approximation of the surface integrals in Eqn. 3.3. The state variables can be simply the volume-average of the left and right cells with respect to face i, denoted by $Q_{pl,i}$ and $Q_{pr,i}$. Notice we use the lower case letter l and r to differentiate the state variables and the cell-average variables. The more sophisticated choice of state variables involves some extrapolation from cells to faces, which we will introduce later. The concept of left and right state comes from **the Riemann problem** [3] and we will not expand here. One can simply understand the concept by relating to the fact that, to compute the face flux, we need both the information from left and right side of the face. One should keep in mind that no matter how sophisticated the construction of the state variables is, given all the cell-average $Q_{p,I}$'s, the left and right state at every face i, $(Q_{pL,i}, Q_{pR,i})$ can be obtained. A final comment about the numerical flux function $\mathcal{F}$. The design of such function is to approximate the face flux such that the numerical solutions of Q_p approaches the real solutions. There can be many choices of such flux functions and the design of numerical flux functions is an important topic of CFD.

Through the spatial discretization in Eqn. 3.2 and 3.3, we have replaced the integrals in the governing equation by a bunch of summations. However, there is still a temporal differentiation in the governing equation, which is not yet fully discretized. To perform the temporal discretization, we approximate the differentiation by finite differences:

$$\Gamma(Q_{p,I})\frac{\partial Q_{p,I}}{\partial t} \approx \Gamma(Q^n_{p,I})\frac{Q^{n+1}_{p,I} - Q^n_{p,I}}{\Delta t} \tag{3.4}$$

Notice that we add the superscript n and indicate the discretized solution for cell I at the n^{th} time moment as $Q^n_{p,I}$. Δt is referred to as the **time step** of the numerical solution. We assume the time step size is constant although technically it can be changing between the n^{th} and the $(n+1)^{th}$ time moment. We point out that, by applying the temporal discretization (Eqn. 3.4), the numerical solution will consists of a time sequence $\{Q^0_{p,I}, Q^1_{p,I}, Q^2_{p,I}, \cdots\}$ for each cell I. The **initial condition** for each cell, $Q^0_{p,I}$ is given, and based on the initial condition, the temporal evolution of primitive variables in every cell will be solved incrementally by the numerical scheme.

The time discretization given by Eqn. 3.4 is of only first order in terms of finite difference approximation. A second order approximation can be written as:

$$\Gamma(Q_{p,I})\frac{\partial Q_{p,I}}{\partial t} \approx \Gamma(Q^n_{p,I})\frac{2Q^{n+1}_{p,I} - 3Q^n_{p,I} + Q^{n-1}_{p,I}}{2\Delta t} \tag{3.5}$$

For the second order approximation, Q_p at time moment n and $(n-1)$ is given to compute Q_p at time moment $(n+1)$. Accordingly, we need two initial conditions, we typically assume $Q^1_{p,I} = Q^1_{p,I}$ to proceed to solve $Q^2_{p,I}$, $Q^3_{p,I}$, etc. In a more general context, we can apply the m^{th} order finite difference to the temporal discretization as:

$$\Gamma(Q_{p,I})\frac{\partial Q_{p,I}}{\partial t} \approx \Gamma(Q^n_{p,I}) \sum_{j=n-m+1}^{1} g_j Q^{n+j}_{p,I}$$

The coefficients g_j's are referred to as the **Gear's coefficients** for temporal discretization.

So far we only applied the temporal discretization to Eqn. 3.2a, now we want to add the superscript to the other terms in Eqn. 3.2b and Eqn. 3.3 to complete the spatial and temporal discretization. In the **explicit time discretization**, we simply evaluate the terms in Eqn. 3.2b and Eqn. 3.3 at the current time moment, i.e., time moment n, resulting in the fully discretized equation as:

$$\begin{aligned}\Gamma(Q^n_{p,I})\frac{2Q^{n+1}_{p,I} - 3Q^n_{p,I} + Q^{n-1}_{p,I}}{2\Delta t}\Omega_I + \sum_{i\in I_{sf}} \mathcal{F}(Q^n_{pL,i}, Q^n_{pR,i}, n_{Ii})\sigma_i - \\ \sum_{i\in I_{sf}} F_{vn}(Q^n_{p,i}, (\nabla \otimes Q_p)^n_i, n_{Ii})\sigma_i = S(Q^n_{p,I})\Omega_I\end{aligned} \tag{3.6}$$

In Eqn. 3.6, the **known** variables will be the primitive variables in each cell I, at time n and $n-1$, i.e., $Q^n_{p,I}$ and $Q^{n-1}_{p,I}$. The face-related variables (with subscript i) can be interpolated/extrapolated based on the cell primitive variables. The **unknown** variables we aim to solve is the cell primitive variables at the next time step $n+1$, $Q^{n+1}_{p,I}$. We can manoeuvre Eqn. 3.6 such that all the unknowns are put on the LHS and all the knowns are put on the RHS:

$$\begin{aligned}&\mathcal{L}(\vec{Q}^n_p)\Delta\vec{Q}^n_p = \mathcal{R}(\vec{Q}^n_p),\\ &\text{where } \mathcal{L}_{II} = \frac{\Gamma(Q^n_{p,I})\Omega_I}{\Delta t},\ \mathcal{L}_{IJ} = 0\ (I \neq J),\ \Delta\vec{Q}^n_p = \vec{Q}^{n+1}_p - \vec{Q}^n_p,\ \text{and}\\ &\mathcal{R}_I = -\left(-\frac{\Gamma(Q^n_{p,I})\Delta Q^{n-1}_{p,I}\Omega_I}{\Delta t} + \sum_{i\in I_{sf}}\left[\mathcal{F}(Q^n_{pL,i}, Q^n_{pR,i}, n_{Ii}) - F_{vn}(Q^n_{p,i}, (\nabla\otimes Q_p)^n_i, n_{Ii})\right]\sigma_i\right)\\ &+ S(Q^n_{p,I})\Omega_I\end{aligned} \tag{3.7}$$

Here, the arrow on the variable, e.g., Q^n_p indicates a vector containing Q^n_p for all the cells, and therefore, the subscript I is dropped. For example, if the number of cells is 20, the vector $\vec{Q}^n_p$ is a column vector with have 20 components: $[Q^n_{p,1}, \cdots Q^n_{p,20}]^T$, and each component itself is a vector. For example, the 10^{th} component is $[p_{10}, V_{10}, T_{10}]^T$, which is the pressure, velocity and temperature for the 10^{th} cell. In this case, the unknown vector in Eqn. 3.7, $\vec{Q}^n_p$, would have a length of 100 (in three dimension). To solve this vector, we are basically dealing with

a linear system where a matrix $\mathcal{L}$, multiples the unknown vector $\vec{Q}_p^n$, equals to another vector $\mathcal{R}$. Again, assume $\vec{Q}_p^n$ has a length of 100 as an example, the matrix $\mathcal{L}$ would be 100×100 and the RHS vector $\mathcal{R}$ would be 100×1. We refer to $\mathcal{L}$ as the **left-hand-side matrix**, or **left-hand-side operator**, and refer to $\mathcal{R}$ as the **right-side-side residual**, or simply the **residual**. As indicated in Eqn. 3.7, the LHS matrix in this case is a block diagonal matrix, while the residual is a complex function of $\vec{Q}_p^n$ and $\vec{Q}_p^{n-1}$. However, since the LHS matrix is block diagonal, we can simply invert $\mathcal{L}$ to solve $\Delta\vec{Q}_p^n$:

$$\Delta\vec{Q}_p^n = \mathcal{L}^{-1}(\vec{Q}_p^n)\mathcal{R}(\vec{Q}_p^n)$$

We can see that the original governing equations Eqn. 3.1, which are integral equations, have become a system of linear equations, after the spatial and temporal discretization.

To close this section, let's briefly revisit the data structure and discuss how information regarding Eqn. 3.7 is stored. We add the following attributes to the `cell` type:

```
type cell
   ...
   type(vector) :: qv, dqv
   type(matrix) :: dm
   type(vector) :: res
   type(vector) :: qvn(:)
end type cell
```

where the type `vector` is another derived data type:

```
type vector
   real :: v(neq)
end type vector
```

The `vector` has one attribute which is a 1D array of floating point numbers. The variable `neq` is a global variable indicating the number of governing equations in Eqn. 3.1. E.g., `neq = 5` in 3D and `neq = 4` in 2D, etc. The type `matrix` is also a derived data type:

```
type matrix
   real :: e(neq, neq)
end type matrix
```

The `matrix` type has one attribute which is a 2D array of floating numbers. In the added attributes of `cell`, we will use `qv` to store the updated primitive variable $Q_{p,I}^{n+1}$, `dqv` to store the increment $\Delta Q_{p,I}^n = Q_{p,I}^{n+1} - Q_{p,I}^n$, and `qvn` to store the primitive variables in the previous time steps, $Q_{p,I}^n$ and $Q_{p,I}^{n-1}$. Notice that `qvn` is an array of `vector`. Its length will depend on the order of the temporal discretization. In the case of second order as in Eqn. 3.6, we need two previous time moments and the length of `qvn` will be two. Finally, the attribute `dm` stores the diagonal terms of the LHS matrix $\mathcal{L}_{II}$ and the attribute `res` stores the residual for the cell, $\mathcal{R}_I$.

3.2 Gradient Computation

From now on, our discussion will be built on the key equation 3.7 which is that a LHS matrix $\mathcal{L}$ multiplies the unknown vector ΔQ_p equals to the residual $\mathcal{R}$. This chapter will be devoted to the computation of gradients for both cells and faces. The gradients will be used in the residual when calculating the left and right states, as well as in the viscous flux. Specially, we are aiming at computing the following matrix:

$$\nabla \otimes Q_p = \begin{bmatrix} \partial_x \\ \partial_y \\ \partial_z \end{bmatrix} \begin{bmatrix} p & u & v & w & T \end{bmatrix} = \begin{bmatrix} \partial_x p & \partial_x u & \partial_x v & \partial_x w & \partial_x T \\ \partial_y p & \partial_y u & \partial_y v & \partial_y w & \partial_y T \\ \partial_z p & \partial_z u & \partial_z v & \partial_z w & \partial_z T \end{bmatrix}$$

Note that we use the short-hand notation $\partial_x p$ for $\partial p / \partial x$, etc. Also, (u, v, w) are the (x, y, z) components of the velocity vector V. The gradient matrix $\nabla \otimes Q_p$ in this case is a 3×5 matrix. We will be computing such a matrix for each cell and face given the primitive variable at each cell $Q_{p,I}$.

Cell Gradient, Method 1: Least Mean Square

Let's consider the pentagon cell I shown in Fig. 3.1. To include more generality, we assume cell I has 5 triangle neighboring cells, indexed as $I1, \cdots, I5$. The centroid of the cells are labeled with C. We also draw 5 vectors from the centroid of cell I to its neighboring cells' centroid as $d_{I1}, \cdots, d_{I5}$. The objective is to compute the gradient matrix at cell I, given that the primitive variables Q_p are known at cell $I, I1, \cdots, I5$.

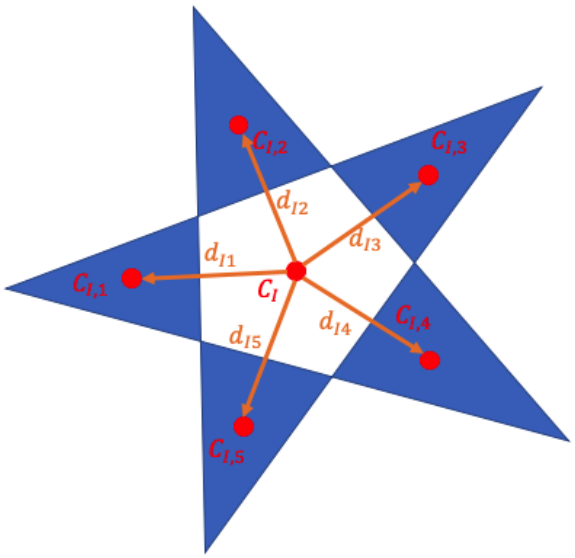

Figure 3.1: Illustration of cell gradient computation.

Let's take the gradient of pressure for example. Knowing $p_I, p_{I1}, \cdots, p_{I5}$, we can write down the following approximation:

$$\begin{bmatrix} \Delta x_{I1} & \Delta y_{I1} & \Delta z_{I1} \\ \Delta x_{I2} & \Delta y_{I2} & \Delta z_{I2} \\ \Delta x_{I3} & \Delta y_{I3} & \Delta z_{I3} \\ \Delta x_{I4} & \Delta y_{I4} & \Delta z_{I4} \\ \Delta x_{I5} & \Delta y_{I5} & \Delta z_{I5} \end{bmatrix} \begin{bmatrix} \partial_x p \\ \partial_y p \\ \partial_z p \end{bmatrix} \approx \begin{bmatrix} \Delta p_{I1} \\ \Delta p_{I2} \\ \Delta p_{I3} \\ \Delta p_{I4} \\ \Delta p_{I5} \end{bmatrix} \tag{3.8}$$

where $\begin{bmatrix} \Delta x_{IJ} & \Delta y_{IJ} & \Delta z_{IJ} \end{bmatrix} = d_{IJ}^T, J = 1, 2, \cdots, 5,$ and

$\Delta p_{IJ} = p_{IJ} - p_I, J = 1, 2, \cdots, 5$

We can understand the task of computing the gradient of p as the follows: find the gradient of p such that the difference (error) between the LHS and RHS of Eqn. 3.8 is minimized. This is a **Least Mean Square** (LMS) problem. That is, find the best vector x that minimizes the approximation $Ax \approx b$. In the case of Eqn. 3.8, x is a 3×1 vector, A is a 5×3 matrix and b is a 5×1 vector. The solution to the LMS problem is to find the **pseudo inverse** of matrix A, A^+, such that $x = A^+ b$ produces the optimal x minimizing the approximation error. In the case of Eqn. 3.8, we can express the solution as:

$$\begin{bmatrix} \partial_x p \\ \partial_y p \\ \partial_z p \end{bmatrix} = \begin{bmatrix} s_{x1} & s_{x2} & s_{x3} & s_{x4} & s_{x5} \\ s_{y1} & s_{y2} & s_{y3} & s_{y4} & s_{y5} \\ s_{z1} & s_{z2} & s_{z3} & s_{z4} & s_{z5} \end{bmatrix} \begin{bmatrix} \Delta p_{I1} \\ \Delta p_{I2} \\ \Delta p_{I3} \\ \Delta p_{I4} \\ \Delta p_{I5} \end{bmatrix} \tag{3.9}$$

where the matrix with components s's is the pseudo inverse of the matrix on the LHS on Eqn. 3.8. For reasons to be illustrated, we shall refer to this matrix as the **SVD** matrix. The subscripts of s's can be understood as some contributing weights. For example, s_{y3} is the contributing weights of the **third** neighboring cell to the **y** direction of the gradients, etc. We note that the SVD matrix is only determined by the directional vectors d_{IJ} in Fig. 3.1 and Eqn. 3.9 can be applied to any primitive variables besides p.

Now we shall present a general expression for computing the gradients of all primitive variables, given that the pseudo inverse, i.e., the SVD matrix, is known, as follows:

$$\begin{bmatrix} \partial_x Q_p & \partial_y Q_p & \partial_z Q_p \end{bmatrix} = \begin{bmatrix} \Delta Q_{p,I1} & \Delta Q_{p,I2} & \Delta Q_{p,I3} & \Delta Q_{p,I4} & \Delta Q_{p,I5} \end{bmatrix} \begin{bmatrix} s_{x1} & s_{y1} & s_{z1} \\ s_{x2} & s_{y2} & s_{z2} \\ s_{x3} & s_{y3} & s_{z3} \\ s_{x4} & s_{y4} & s_{z4} \\ s_{x5} & s_{y5} & s_{z5} \end{bmatrix} \tag{3.10}$$

where $\partial_x Q_p = \begin{bmatrix} \partial_x p & \partial_x u & \partial_x v & \partial_x w & \partial_x T \end{bmatrix}^T$, same for y and z, and

$\Delta Q_{p,IJ} = \begin{bmatrix} \Delta p_{IJ} & \Delta u_{IJ} & \Delta v_{IJ} & \Delta w_{IJ} & \Delta T_{IJ} \end{bmatrix}^T, J = 1, 2, \cdots, 5$

Note that we transposed the SVD matrix to match its contributing weights. We now briefly discuss how the SVD can be obtained by solving the LMS problem Eqn. 3.8. The task is to find the pseudo inverse the matrix containing all the directional vectors d_{IJ}. Finding the pseudo inverse, or solving the LMS problem, falls in the knowledge of **Numerical Linear Algebra** [4]. The most general way of solving a LMS problem is via the **Singular Value**

Decomposition (SVD). The numerical implementation of SVD is somewhat involved and is the "crown" of numerical linear algebra [5]. We shall not discuss the algorithm of SVD here. Most of the times, the SVD algorithm can be viewed as a black-box subroutine whose input is the A matrix and output is the pseudo inverse A^+. Technically, the pseudo inverse matrix is not the SVD itself, but we still refer to the pseudo inverse matrix as the SVD matrix, which will be the matrix with components s's in Eqn. 3.10. Finally, we point out that new attributes need to be added to the type `cell` for the computation of Eqn. 3.10:

```
type cell
   ...
   real, pointer :: gradient(:,:)
   real, pointer :: svd(:,:)
end type cell
```

Note that both matrices of `gradient` and `svd` has undefined lengths as they depend on the dimension of the problem, etc. In the case of Eqn. 3.10, the `gradient` has a dimension of 5×3 and `svd` has a dimension of 5×3.

Constructing Neighbor Structure

With Eqn. 3.10 we can compute the gradients at cell I given the primitive variables of the neighboring cells of cell I. We need to construct data structures to store the connectivity information between cell I and its neighboring cells. We shall achieve this via the `left_cell` and the `right_cell` pointers at the faces. First, we add the attribute in the `cell` type:

```
type cell
   ...
   type(neighbour), pointer :: sface(:)
   type(neighbour_cell), pointer :: scell(:)
end type cell
```

The data type `neighbour` and `neighbour_cell` are the nested data type:

```
type neighbour
   type(face), pointer :: to_face
end type neighbour

type neighbour_cell
   type(face), pointer :: to_cell
end type neighbour_cell
```

The nested data structure is to indicate that the pointers are already allocated. The attribute `sface(:)` is a list of face pointers (already allocated) referencing to the neighboring faces of the current cell. The attribute `scell(:)` is a list of cell pointers (already allocated) referencing to the neighboring cells of the current cell. The list `sface(:)` can be obtained in the following way:

```
loop faces
   for each face cf:
```

```
      lc => cf % left_cell
      rc => cf % right_cell
      append cf to the sface(:) of lc
      append cf to the sface(:) of rc
   end loop
```

With `sface(:)`, the list `scell(:)` can be readily obtained:

```
   loop cells
      for each cell cc
      loop cc % sface
         for each face, cf, in cc % face(:)
         lc => cf % left_cell
         rc => cf % right_cell
         if lc is cc
            append rc to the scell(:) of cc
         else
            append lc to the scell(:) to cc
         end if
      end loop
   end loop
```

With these connectivity structures, we can design the pipeline of gradients computation as follows:

1. For each cell, find the directional vectors d_{IJ} from its neighboring cells as in Fig. 3.1.
2. Based on the directional vectors, calculate the SVD matrix for each cell.
3. For each cell, gather the primitive variables from its neighboring cells.
4. Compute the gradients according to Eqn. 3.10.

Cell Gradient, Method 2: Gauss-Green Approach

We introduce the second method of calculating the cell gradient. The gradient of any scalar variable, taking pressure as an example, can be expressed using the Gauss's theorem (refer back to Eqn. 2.6a) as:

$$\int_{\Omega} \nabla p \, d\Omega = \int_{\partial\Omega} p \, n \, d\sigma$$

Note here Ω is the cell as a control volume and $\partial\Omega$ is the surrounding faces (`sface(:)`) of the cell as the boundary of the control volume. We can approximation the integrations as:

$$(\nabla p)_I \Omega_I = \sum_{i \in I_{sf}} p_i n_{Ii} \sigma_i$$

where n_{Ii} is the face norm of face i pointing **out of** cell I. Therefore, we can simply compute the gradient of cell I by:

$$\nabla p = \sum_{i \in I_{sf}} p_i n_{Ii} \frac{\sigma_i}{\Omega_I} \tag{3.11}$$

This method of computing gradient can be applied all the primitive variables of cell I. There are two following issues with Eqn. 3.11. First, the averaged primitive variables at each surrounding face i, $Q_{p,i}$, need to be precomputed, and second, the normal vector n_{Ii} needs to be pointing out of cell I to ensure its correct direction.

Let's tackle the first issue. We can define the **face average** primitive variables at each face i, $Q_{p,i}$, as the average of the **nodal primitive variables** for face i, i.e.:

$$Q_{p,i} = \frac{1}{|i_{f2n}|} \sum_{N \in i_{f2n}} Q_{p,N} \tag{3.12}$$

Here, i_{f2n} is the set of node indices which is associated with face i (recall that we already have this information as the **face-node connectivity** when we generate the mesh back in chapter 1). $Q_{p,N}$ is the primitive variables at node N. We only explicit store the primitive variables at nodes and compute the face primitive variables with Eqn. 3.12 when needed. Thus, we add the attribute to the `node` type:

```
type node
   ...
   type(vector) :: qv
end type node
```

The question now becomes how to compute the nodal primitive variables $Q_{p,N}$. This is achieved by a **weighted inverse distance** interpolation based on the cell primitive variables $Q_{p,I}$. Let's redraw the mesh for partition 1 in Fig. 1.4 below as Fig. 3.2.

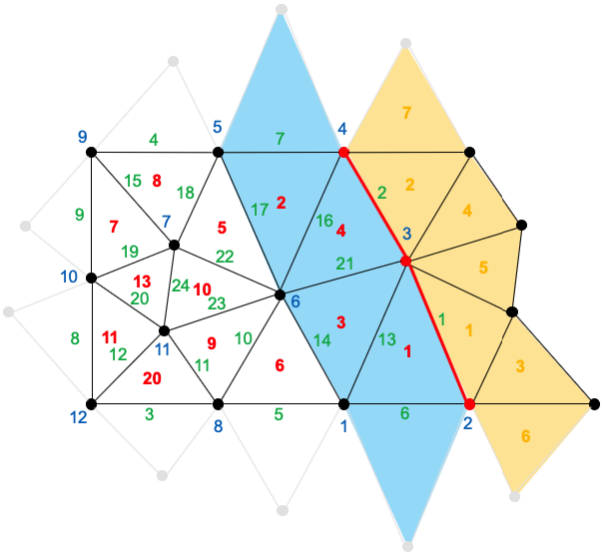

Figure 3.2: Illustration of nodal interpolation by the mesh in one of the partitions.

We have 12 nodes in this partition labeled blue. They are connected to cells via the cell-node connectivity stored in the structure `c2n`. Let's review the types of cells that connect to the nodes:

1. **Interior cells**: labeled by red from 1 to 20. They are stored in the `cells(:)` array.

2. **Partition (ghost) cells**: labeled by orange from 1 to 7. They are stored in the `pcell(:)` array in the structure `itf` (named as `interf`).
3. **Boundary (ghost) cells**: there is no label. They are the `right_cell` of the boundary faces labeled from 3 to 9.

All of the cells have the attributes `c2n(:)` that stores the indices of nodes (1 to 20) that associated with the cell. Therefore, we can find the distance between any cell center to its associated nodal coordinates. We add the `weight` attribute to the type `cell`:

```
type cell
   ...
   real, pointer :: weight(:)
end type cell
```

The length of the `weight` will be equal to the length of the integer pointer `c2n`. We can assign the weights by:

```
loop cells
   for each cell cc
   loop c2n of cc
      find node by nodes(cc(c2n(:)))
      for each node cn
      compute inverse distance between cc % centp to cn % xyz
      and save it to cc % weight
   end loop

   normalize cc % weight
end loop

looping parition cells, same
looping boundary cells, same
```

and the nodal interpolation can be done by:

```
nodes % qv = zero
loop cells
   for each cell cc
   loop c2n of cc
      find node by nodes(cc(c2n(:)))
      for each node cn, get the weights for this node
      cn % qv = cn % qv + cc % qv * this weight
   end loop
end loop

looping parition cells, same
looping boundary cells, same
```

Once the nodal primitive variables $Q_{p,N}$ are computed, we can compute the face primitive variables $Q_{p,i}$ (faces are labeled by green in Fig. 3.2) by Eqn. 3.12, which can be used to

compute the cell gradient in Eqn. 3.11. Notice that $Q_{p,i}$ will also be used when computing the viscous flux F_{vn} in Eqn. 3.7.

Now let's tackle the issue of the sign of the normal vector n_{Ii}. The face norm can be readily computed knowing the nodal coordinates on the face. But the direction of this norm is still depending. We mandate that the **face norm** always points from its `left_cell` to its `right_cell`. This face norm is stored in the type `face`, denoted as n_i. We add the `vecn` attributes to the type `face` to store the face norm:

```
type face
   ...
   real :: vecn(ndim)  ! ndim is number of dimensions
end type face
```

We need not to separately store face norms n_{Ii} for each cell I, as we already know n_i, and n_i and n_{Ii} only differ by the sign. We note that when computing the gradient via Eqn. 3.11, the gradient becomes a summation. We can elegantly deal with the sign of face norms by choosing to loop over faces but not the cells, as follows:

```
subroutine cell_gradient
   cells % gradient = zero
   loop faces
      for each face cf
      find the associated nodes by cf % f2n
      average nodal primitives as face primitives qvf
      cl => cf % left_cell
      cr => cf % right_cell
      a = cf % area

      loop i = 1, 2, 3
         cl % gradient(:,i) = cl % gradient(:,i) +
                              qvf * cf % vecn(i) * a
         cr % gradient(:,i) = cr % gradient(:,i) -
                              qvf * cf % vecn(i) * a
      end loop
   end loop
   cells % gradient = cells % gradient / cells % vol
end subroutine cell_gradient
```

Note that for the `left_cell` we use the plus sign in the summation, but for the `right_cell` we use the minus sign. This is because for the left cell, n_i and n_{Ii} have the same sign, as n_i points **from left cell to right cell** and n_{Ii} points **out of cell**. This is the opposite for the right cell.

Face Gradient

At the last of this section, we discuss the computation of gradient of primitive variables at each face i, which will be used in the computation of viscous flux F_{vn} in Eqn. 3.7. We take a similar approach as the LMS method in the cell gradient computation. First we compute for each face i, three directional vectors as shown in Fig. 3.3. This is illustrated in 3D as an example, and in 2D, we only need to construct two directional vectors for each face i. If the face is quadrilateral, the first two directional vectors would be a "cross" connecting the diagonal nodes belonging to face i. The third directional vector connects the left and right cell of face i (indexed as il and ir). If the face is triangle, the first two directional vectors are two edges of the triangle (randomly chosen), and the third directional vector still connects the left and right cell.

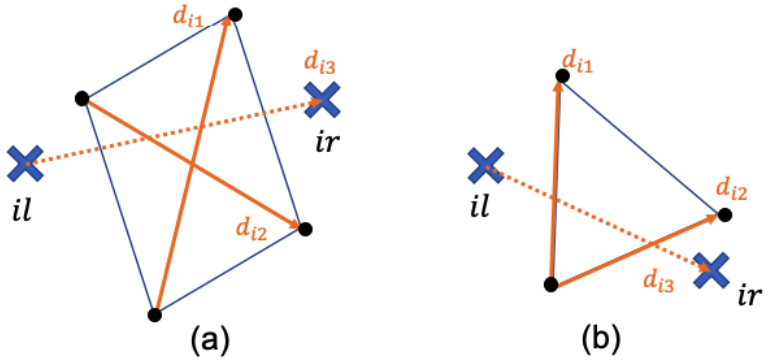

Figure 3.3: Illustration of face gradient computations. (a) Quadrilateral face and (b) Triangle face.

Now similar to Eqn. 3.8, we can write down the following approximation (again, take the pressure gradient as an example):

$$\begin{bmatrix} \Delta x_{i1} & \Delta y_{i1} & \Delta z_{i1} \\ \Delta x_{i2} & \Delta y_{i2} & \Delta z_{i2} \\ \Delta x_{i3} & \Delta y_{i3} & \Delta z_{i3} \end{bmatrix} \begin{bmatrix} \partial_x p \\ \partial_y p \\ \partial_z p \end{bmatrix} \approx \begin{bmatrix} \Delta p_{i1} \\ \Delta p_{i2} \\ \Delta p_{i3} \end{bmatrix} \tag{3.13}$$
$$\text{where} \quad \begin{bmatrix} \Delta x_{ij} & \Delta y_{ij} & \Delta z_{ij} \end{bmatrix} = d_{ij}^T, j = 1, 2, 3$$

Here the term Δp_{ij} are taken to be the pressure difference between the starting and ending point of the directional vectors. For example, $\Delta p_{i3} = p_{ir} - p_{il}$, etc. Note that the nodal primitive variables should already been computed by the weighted inverse distance average algorithm discussed previously. We note that Eqn. **??** is not an over-determined system and can be solved by a direct inversion:

$$\begin{bmatrix} \partial_x p \\ \partial_y p \\ \partial_z p \end{bmatrix} = \begin{bmatrix} \Delta x_{i1} & \Delta y_{i1} & \Delta z_{i1} \\ \Delta x_{i2} & \Delta y_{i2} & \Delta z_{i2} \\ \Delta x_{i3} & \Delta y_{i3} & \Delta z_{i3} \end{bmatrix}^{-1} \begin{bmatrix} \Delta p_{i1} \\ \Delta p_{i2} \\ \Delta p_{i3} \end{bmatrix} \tag{3.14}$$

We shall precompute the matrix (inverted) on the RHS of Eqn. 3.14 and store it in the type `face` by the attributes `avec`:

```
type face
```

```
    ...
    real :: avec(ndim,ndim)
end type face
```

By doing this, we only need to compute the matrix multiplication in Eqn. 3.14 to obtain the gradient at face i.

3.3 Face Flux Computation

3.4 Implicit Temporal Discretization

3.5 Boundary Conditions

3.6 Preconditioning: Implementation

3.7 Solving Linear Systems

Chapter 4: Awkward Level-Set GEMS

4.1 The Level-Set Function

4.2 Hamilton-Jacobi Equations

4.3 Ghost Fluid Method

4.4 Lagrangian Particle Tracking*

Chapter 5: Benchmark Tests

5.1 One-Dimensional Euler Equation

5.2 Propagation of Vortex

5.3 Flow Around a Cylinder

5.4 Decay of Isotropic Turbulence*

Reference

[1] R. L. Panton, *Incompressible flow*. John Wiley & Sons, 2013.

[2] D. Li and C. L. Merkle, "A unified framework for incompressible and compressible fluid flows," *Journal of Hydrodynamics, Ser. B*, vol. 18, no. 3, pp. 113–119, 2006.

[3] E. F. Toro, *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer Science & Business Media, 2013.

[4] L. N. Trefethen and D. Bau III, *Numerical linear algebra*. Siam, 1997, vol. 50.

[5] G. Golub and C. Van Loan, "Matrix computations 4th edition the johns hopkins university press," *Baltimore, MD*, 2013.