Immersed Boundary Method Implementation Notes

Pedro de Almeida Secchi

August 1, 2025

Contents

Chapter 1

Introduction

This document includes implementation advice for Immersed Boundary Method (IBM) solution codes for Computational Fluid Dynamics applications. I have learned that these implementation details were crucial for the success of a "newborn" CFD code through trial and error during my PhD, and hereby leave them to you, the reader.

Each chapter is dedicated to a facet of the IBM, and describes how certain numerical methods can be written for a "baseline" implementation and a working CFD code, including:

1. Geometry handling with triangulations;
2. Octree/quadtree meshing; and
3. Operator definitions.

Good luck with your programming!

Chapter 2

Geometry Handling

Geometry handling with the Immersed Boundary Method dramatically favours triangulations over analytical or NURBS parameterizations, since triangulations have easier algorithms for point-in-polygon queries and point projections on surfaces. We will use both in the next chapter for meshing purposes, and they will be described in the following sections.

Each triangulation should be described by a struct containing a point cloud and simplex corner indices. Not much more is expected for the algorithms presented below, but a matrix format is suggested for both.

2.1 Point-in-polygon queries

To perform point-in-polygon queries, a bounding box tree data structure is suggested. The main point is to separate the simplices of a triangulation into branches of a tree by splitting the domain with a plane. Each simplex is allocated to the branch corresponding to the side of the plane its center point occupies.

Starting from a triangulation, one may define which Cartesian direction the splitting plane will be normal to based on which dimension results in a clearer separation between simplex groups. This clarity may be measured by a score S:

$$S = min\{1 - N_l/N, 1 - N_r/N\} \tag{2.1}$$

Where N_l is the number of simplices to the "left" of the splitting plane, N_r is the number of simplices to the "right", and $N = N_l + N_r$.

The bounding box surrounding a set of simplices should be aligned with the Cartesian axes, which will ease its geometric parameterization. Also, a size threshold should, for optimal performance, be established to stop the triangulation splitting once N is below a given number of simplices.

The queries may be done by ray tracing – *i. e.* by checking if the number of intersections between the surface and a line linking the query point to an exterior reference is odd. The main optimization – which makes this verification $\mathcal{O}(M \times log(N))$, if N is the number of simplices and M is the number of query

points – is done by ensuring that only simplices in a bounding box intersected by the ray need to be checked. One may, thus, go down the tree exploring only branches which have their bounding boxes transfixed by the ray, and making analytical intersection checks once a leaf is reached.

2.1.1 Bounding box/ray intersection

To check whether a ray with extremes $\vec{p}_1$ and $\vec{p}_2$ intersects a bounding box given by its origin $\vec{r}_0$ and widths $\vec{w}$, one may find non-dimensional coordinates ξ within the ray with which it intersects the planes defining the limits of the box. This can be done with the following relations:

$$\begin{aligned} \vec{\xi} &= (\vec{r}_0 - \vec{p}_1) \oslash (\vec{p}_2 - \vec{p}_1) \\ \vec{\xi} &= (\vec{r}_0 + \vec{w} - \vec{p}_1) \oslash (\vec{p}_2 - \vec{p}_1) \end{aligned} \tag{2.2}$$

For the "fore" and "aft" faces of the box along each dimension, respectively. $\oslash$ denotes Hadamard (element-wise) division.

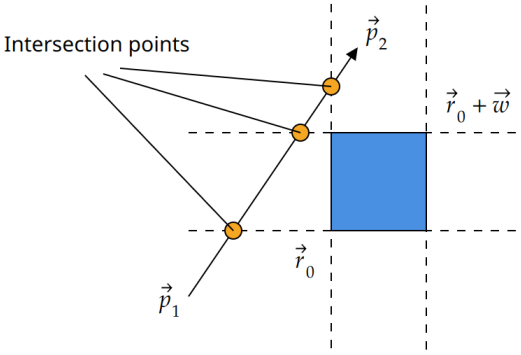

Figure 2.1: Illustration showing the intersection points represented by one-dimensional parameter ξ.

Subsequently, the values in vector $\vec{\xi}$ may be clamped to a range between 0 and 1. If, for any value ξ_i, one observes that the point $\vec{r}(\xi_i) = \vec{p}_1 + (\vec{p}_2 - \vec{p}_1) \times \xi_i$ lies on the bounding box, then the ray intersects the bounding box.

Since this is a heuristic method that serves only to check whether a ray is **likely** to intersect the box, margins of $\sqrt{\epsilon}$ may be added to this point-on-box check for robustness with small boxes and fine meshes.

2.1.2 Analytical simplex/ray intersection

In a bounding box tree leaf, one must evaluate the number of intersections between simplices in the triangulation and the ray. We will hereby exemplify how this can be done in 3D, and the restriction to 2D is left for the reader.

Assume that the ray connects points $\vec{p}_1$ and $\vec{p}_2$, and that the simplex being checked is defined by vertices $\vec{r}_1$, $\vec{r}_2$ and $\vec{r}_3$. The vertices can also be described by an "origin" $\vec{r}_1$ and vectors $\vec{u}, \vec{v}$, such that:

$$\begin{aligned} \vec{u} &= \vec{r}_2 - \vec{r}_1 \\ \vec{v} &= \vec{r}_3 - \vec{r}_1 \end{aligned} \tag{2.3}$$

If we define vector $\vec{w} = \vec{p}_2 - \vec{p}_1$ as well, we may build a vector basis with conversion matrix M:

$$M = \begin{bmatrix} u & v & w \end{bmatrix} \tag{2.4}$$

One may transfer points $\vec{p}_1$ and $\vec{p}_2$ to the given basis and have the origin of their coordinate system shifted to $\vec{r}_1$ by using:

$$\begin{aligned} \vec{\xi}_1 &= M^\dagger \left(\vec{p}_1 - \vec{r}_1 \right) \\ \vec{\xi}_2 &= M^\dagger \left(\vec{p}_2 - \vec{r}_1 \right) \end{aligned} \tag{2.5}$$

If $()^\dagger$ indicates a pseudo-inverse.
In this coordinate system, we may note that:

1. Coordinate $\xi^{(3)}$ is now positive for a point above the plane defined by the simplex, and negative for a point below it;

2. Coordinates $\xi^{(1)}$ and $\xi^{(2)}$ are such that, if a point lies within the simplex, then $\xi^{(1)} + \xi^{(2)} \leq 1$ and $\xi^{(i)} \geq 0$, $i = 1, 2$; and

3. $\xi_1^{(3)} = \xi_2^{(3)}$, given the definition of basis vector $\vec{w}$.

From these observations, the following criteria ensue for a ray to intersect a simplex:

$$\xi_1^{(i)} \geq 0, i = 1, 2 \tag{2.6}$$

$$\xi_1^{(1)} + \xi_1^{(2)} \leq 1 \tag{2.7}$$

$$\xi_1^{(3)} \times \xi_2^{(3)} \leq 0 \tag{2.8}$$

Note that the third condition ensures that the ray's extremes lie on opposite sides of the plane.

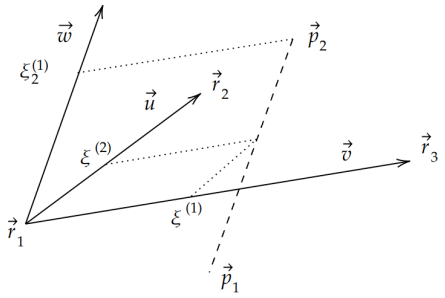

Figure 2.2: Coordinates and points involved in ray/simplex intersection.

2.2 Projection on surface

To obtain the projection $\vec{p}$ of a point $\vec{r}$ on a triangulated surface, we will use the aforementioned bounding box tree data structure to obtain $\mathcal{O}(log(N))$ efficiency.

The algorithm is rather trivial. We begin with a guess $\hat{p}$ for the projection of $\vec{r}$ on the surface and represent the distance between $\vec{r}$ and $\hat{p}$ by $\hat{d}$.

At each three node, the minimum distance between $\vec{r}$ and anything in the bounding box of each branch will be obtained. If said minimum distance is lower than $\hat{d}$, then one must continue going down the branch in a recursive search and update $\hat{p}$ and $\hat{d}$ once the search is done. If both branches stemming from a node show potential, then it is beneficial to follow the branch with a smallest minimum distance to $\vec{r}$. Instead, if none of the branches shows a minimum distance to the content of the bounding box lower than $\hat{d}$, then $\hat{p}, \hat{d}$ must not be updated.

Once all branches with potential are explored, $\vec{p} = \hat{p}$.

Whenever a leaf is reached, the projection $\vec{p}$ must be obtained analytically by projecting $\vec{r}$ on all simplices in the triangulation contained by the leaf's bounding box, and finding which projection is closest.

2.2.1 Minimum distance between a point and a the contents of a bounding box

We will once again use the notation $\vec{r}_0, \vec{w}$ for a bounding box's origin and widths vectors, respectively. The center of the box will be denoted by:

$$\vec{c} = \vec{r}_0 + \vec{w} \tag{2.9}$$

Along each spatial dimension n, we will define vector $\vec{u}$:

$$u_n = \max\{|c_n - r_n| - \frac{w_n}{2}, 0\} \tag{2.10}$$

The minimum distance between point $\vec{r}$ and anything within the bounding box is given by the norm $||\vec{u}||_2$.

2.2.2 Analytical projection of point on simplex

To obtain the exact projection of a point $\vec{r}$ on a simplex given by points $\vec{r}_1$, $\vec{r}_2$ and $\vec{r}_3$, we will once again obtain vectors $\vec{u}$ and $\vec{v}$:

$$\begin{aligned} \vec{u} &= \vec{r}_2 - \vec{r}_1 \\ \vec{v} &= \vec{r}_3 - \vec{r}_1 \end{aligned} \tag{2.11}$$

We will build a coordinate system centered around $\vec{r}_1$ given by conversion matrix M:

$$M = \begin{bmatrix} u & v \end{bmatrix} \tag{2.12}$$

The coordinates of the projection of $\vec{r}$ onto the simplex, in said coordinate system, are given by:

$$\vec{\xi} = M^{\dagger}(\vec{r} - \vec{r}_1) \tag{2.13}$$

To check whether the projection lies on the simplex, one must check the following conditions:

$$\begin{aligned} \xi_1 &\geq 0 \\ \xi_2 &\geq 0 \\ \xi_1 + \xi_2 &\leq 1 \end{aligned} \tag{2.14}$$

If the projection lies on the simplex, one may use:

$$\vec{p} = \vec{r}_1 + M\vec{\xi} \tag{2.15}$$

Instead, if the point lies out of the simplex, one may return as $\vec{p}$ the closest projection of $\vec{r}$ on an edge of the simplex. The projection of a point on an edge is very similar, with the sole exception that a lower number of dimensions must be considered.

Chapter 3

Region Tree Meshing

In this chapter, we will go over the generation of an octree/quadtree mesh and the basic geometric algorithms needed to opearte on said mesh. These include:

1. Cell definition;
2. Cell splitting;
3. Leaf collection;
4. Cell numbering;
5. Cell blanking;
6. k nearest cells;
7. Interpolation along octree mesh cells;
8. Tree balancing;
9. Boundary intersection obtention;
10. Refinement at boundaries and distance fields; and
11. Block clustering.

3.1 Cells

A region (quad or octree) tree cell may be defined with the following data:

- An origin $\vec{r}_0$, a widths vector $\vec{w}$ and a center $\vec{c}$;
- An index, indicating a non-zero value for an interior leaf, zero if an exterior leaf or a non-leaf cell with at least one interior leaf, and -1 if a non-leaf cell with no interior leaves;

- An array of children cells; and
- A "split size" integer s, indicating that the tree will be divided, at each cell, between s^N branches, if N is the number of spatial dimensions.

A cell is a leaf if its array of children is empty.

The cells' indices may be used to orient recursive algorithms. For example, if moving down recursively along a region tree, one may abort the recursion without reaching the leaves at a tree node with index -1, since none of its contained leaves is expected to be within the domain.

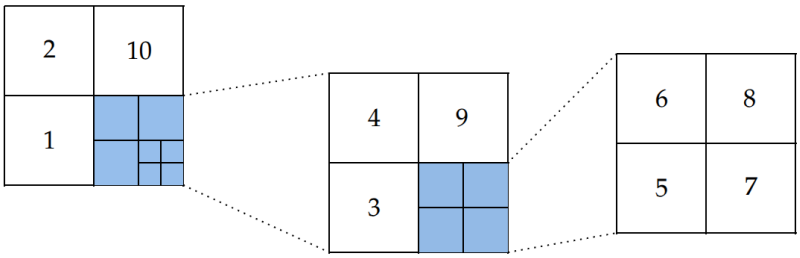

Figure 3.1: Octree with typical index numbering.

3.2 Cell splitting

To obtain another refinement level from a tree leaf, one may fill its array of children with cells of widths vector $\vec{w}/s$ and origins in an s-element range between $\vec{r}_0$ and $\vec{r}_0 + \vec{w}$. In Julia, the corresponding code is:

```
Base.ndims(cell::TreeCell) = length(cell.origin)

cell.children = [
    TreeCell(
        cell.origin .+ delta .* cell.widths, # origin
        cell.widths ./ cell.split_size
    ) for delta in Iterators.product(
        fill(
            LinRange(0.0, 1.0, cell.split_size)[1:(end - 1)],
            ndims(cell)
        )...
    )
]
```

3.3 Leaf collection

To collect the leaves of a region tree to an array, one may define the following, trivial recursive algorithm:

- We start searching from the root;
- If the cell at hand has index $i = 0$ (leaf) or $i < 0$ (non-leaf), nor it, nor its children are within the domain, and an empty vector is returned;
- If the cell in question is a leaf, we return a unit vector containing said leaf. Else, a concatenated array is returned with the collected leaves of all its children.

At the end of the process, the returned array will contain all of the (ordered) leaves of the tree that lie within the domain.

Its useful to also add the option, via a boolean argument, to disregard which cells lie out of the domain and return all tree leaves regardless of their point-in-polygon query status.

3.4 Cell numbering

To define the indices of the cells in a tree, one may define the following recursive process:

- We begin at the root, with $N = 0$, N identifying the total number of leaves within the domain that are contained in the current cell;
- We store the initial value of number N in a temporary variable N_0;
- If the cell at hand is a leaf that lies within the domain, we increment N by 1 and set the leaf's index to $i = N$. Otherwise, $i = 0$;
- If the cell at hand is not a leaf, we run the recursive numbering algorithm for all of its children;
- If, in the end, $N \neq N_0$, then the non-leaf cell must be given index 0, as it contains leaves within the domain. Otherwise, it must be marked with index $i = -1$ (a dead branch lying entirely out of the domain).

Note that N must be passed by reference to each recursive call.

3.5 Cell blanking

Cell blanking consists of marking cells as within or without the domain, and indexing them accordingly. It may be done by initially setting all cells as within the domain (*i. e.* running the routine in Section 3.4 as is).

We may, then, run point-in-polygon queries for all of the leaves' centers and store the results in a "mask" boolean array. The mask can be used to identify which leaves will be given an $i = 0$ index according to the recursive algorithm in 3.4, signaling that they lie outside of the domain.

Both to increase the robustness of the point-in-polygon queries and to improve performance, it is advised to switch the query reference point when going

"downstream" in the tree during a recursive search. When a non-leaf cell is reached, its center may be used as an origin for the ray-tracing performed for its children's point-in-polygon queries, so long as it is far from the boundary (as excessive proximity may lead to robustness issues).

A suitable heuristic for such boundary distance is:

$$d > \frac{||\vec{w}||}{20} \tag{3.1}$$

3.6 k nearest cells

To obtain the k cells of a region tree closest to a given point in space, we may run an insertion sort algorithm and optimize it to $\mathcal{O}(log(N))$ by using the tree structure.

To initialize the insertion sort algorithm, we must define cell and distance vectors of size k and fill them with null pointers and ∞, respectively.

Once again, we start the recursion from the root cell.

We may use the min. distance to bounding box calculation method in Section 2.2.1 to define the minimum distance between the tree cell at hand and the given query point. If said distance is larger than the largest distance to the query point currently stored during the insertion sort process, then the cell may be skipped.

Otherwise, if the cell is a leaf, then it and its center's distance to the query point must be inserted into the storage vectors, in an insertion sort algorithm iteration (*i. e.* by keeping the storage vectors ordered). If it is not a leaf, however, then its children must be explored, as they may contain leaves which are sufficiently close to the query point to be considered for the sorting process.

Once all leaves have been considered, the storage vectors may be returned, as they contain the k cells closest to the query point and their respective distances to it.

3.7 Interpolation along mesh cells

Given a function capable of establishing the k cells closest to a given query point, these cells may have their centers used as an interpolation stencil for the obtention of solution properties at said point. The specific interpolation method is free to be chosen in order to comply with each user's needs, but Sherman's method is recommended for first-order precision, and ridge regression, for second-order.

It is recommended that data structures which enable SIMD parallelization be used as often as possible for this interpolation process, as it will be vital for the performance of the IBM.

3.8 Tree balancing

To balance a region tree means to ensure that no cell i has a neighboring cell j such that:

$$w_k^{(i)} < w_k^{(j)} \times s, k = 1, 2, ...N \tag{3.2}$$

If s is the cell split size, and N is the spatial dimensionality of the mesh. Figure 3.2 exemplifies the effects of balancing a region tree.

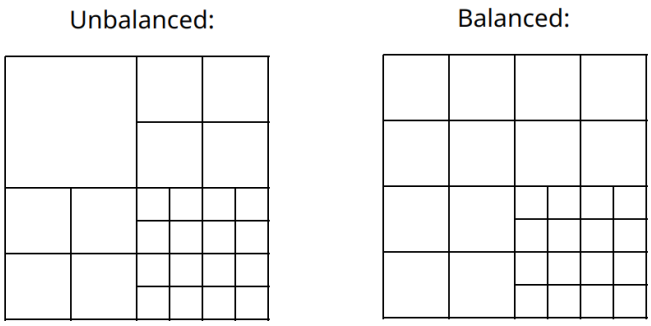

Figure 3.2: Examples of unbalanced and balanced quadtrees.

To find the neighbors of a given reference cell, one may adapt the cell collection algorithm in Section 3.3 to return an empty vector when the cell at hand is distant from the reference cell by more than ϵ, if ϵ is a threshold close to machine zero.

Along each spatial dimension n, we may define vector $\vec{u}$:

$$u_n = \max\{|c_n^{(i)} - c_n^{(j)}| - \frac{w_n^{(i)}}{2} - \frac{w_n^{(j)}}{2}, 0\} \tag{3.3}$$

The distance between cells i and j is given by $||\vec{u}||_2$.

The balancing algorithm is given by the following sequence:

1. Collect all leaves in the tree and define a "mask" vector of booleans, with a value for each leaf, initialized by `false`;
2. For each leaf, find its neighboring leaves in the tree;
3. If any of its neighbors presents $w_k^{(i)} < w_k^{(j)} \times s$ for any spatial dimension k, flag j for splitting;
4. Conversely, if $w_k^{(j)} < w_k^{(i)} \times s$ for any k, flag i for splitting;
5. Once the flagging is done, split all cells with a `true` value stored to their entry in the "mask" vector;

6. Repeat from step 1 until no cells need to be split.

3.9 Boundary intersection

To obtain the intersection between a triangulated domain boundary and an octree mesh, one may perform the recursive leaf collection algorithm in 3.3, but returning an empty vector whenever the considered cell (be it a leaf or not) has its center distant to the boundary by more than $||\vec{w}||/2$.

This distance threshold is equivalent to the radius of the circumcircle or circumsphere of the cell (Figure 3.3). One must, therefore, note that this distance threshold assumes near anisotropy of the octree mesh.

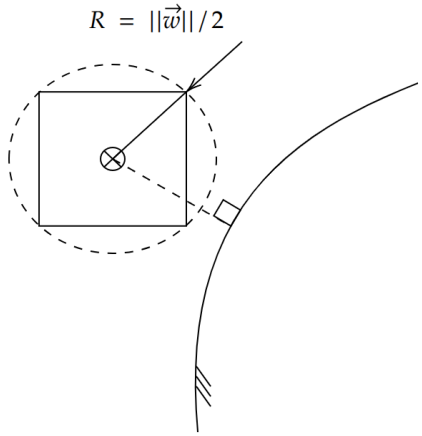

Figure 3.3: Relationship between circumradius of a cell, its distance to a boundary and its potential intersection with the boundary.

3.9.1 Hypercube boundaries

The intersection between a hypercube boundary (a face of the square or cube that envelops the entire region tree mesh) may be simplified when using a programming language that includes slices as a feature. We will exemplify it with Julia's slice notation.

We may represent a hypercube boundary by a tuple between an integer and a boolean variable. The integer represents the spatial dimension the face is perpendicular to, while the boolean indicates if we're dealing with the front or back face of the enveloping hypercube in said dimension. For example, the back face of a square/cube perpendicular to the y-axis ($y = 0$) is represented by `(2, false)`. A face on the plane $x = w_1$, however, would be represented by `(1, true)`. Figure 3.4 exemplifies this notation in 3D.

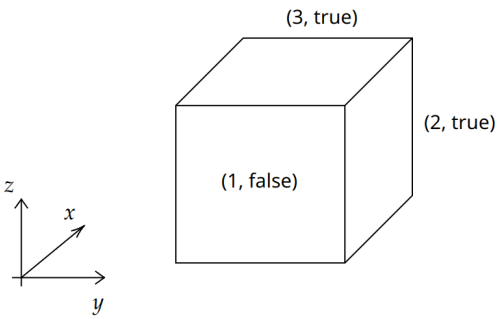

Figure 3.4: Faces of a cube in tuple notation.

We may recursively travel through slices of the array of children for each cell until we're left with children intersecting said face, only. In Julia:

```
isleaf(cell::TreeCell) = (length(cell.children) == 0)

hypercube_boundary(cell::TreeCell, face::Tuple) = (
    isleaf(cell) ?
    [cell] :
    let (dim, front) = face
        [
            hypercube_boundary(ch, face) for ch in selectdim(
                cell.children, dim, (
                    front ?
                    cell.split_size:
                    1
                )
            )
        ]
    end |> vec |> x -> reduce(vcat, x)
)
```

In a two-dimensional example in which we seek to obtain face `(2, false)` of a region tree, we may start at the root and, recursively:

1. Obtain the hypercube boundary `(2, false)` of each cell in `cell.children[:, 1]`;

2. Concatenate the results and return them.

It is trivial to limit the algorithm above to only return cells that are within the domain ($i > 0$).

3.10 Boundary refinement

The refinement at a triangulated boundary may be performed using the intersection detection algorithm described in section 3.9.

We begin by defining the following parameters:

- A characteristic cell length h, which defines the maximum cell width (along any dimension) at the boundary;
- A buffer layer depth d_b, which describes the number of recursive cell splits done to each cell in order to limit the volumetric growth rate of the mesh; and
- A relative detection radius ρ.

We start from a coarse region tree mesh. The following steps are then enacted:

1. Find the tree leaves which intersect the boundary;
2. Split those that have a maximum width $\max\{\vec{w}\} > h \times s^{d_b}$, if s is the cell split size; and
3. Repeat the process until the number of leaves is unchanged.

At the end of the process, the region tree is balanced. Then, d_b recursive splitting steps are followed to all cells, indiscriminately. Figure 3.5 shows how the use of these recursive splits can limit the volumetric growth rate of the mesh.

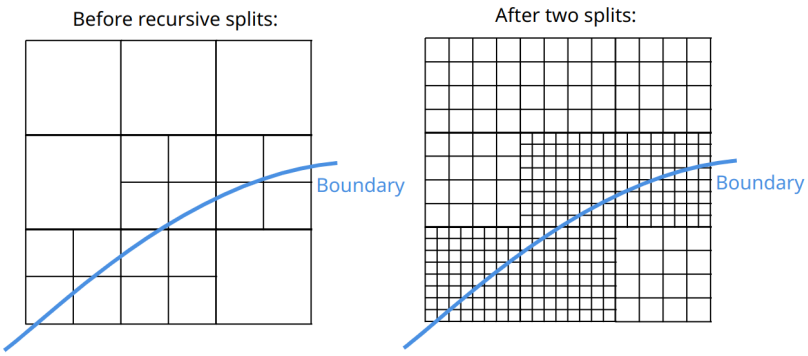

Figure 3.5: Illustration of the mesh before and after the recursive splits.

The relative detection radius is used to tweak the boundary intersection algorithm in 3.9 so that cells with a distance from the boundary larger than ρ times their circumradius are caught as intersecting the boundary:

$$d < \rho \times \frac{||\vec{w}||}{2} \tag{3.4}$$

A value of $\rho > 1$ makes it such that all cells which are reasonably close to the boundary – not just those that directly intersect it – present $\max\{\vec{w}\} \leq h$, limiting the growth ratio of the near-wall cells in the resulting mesh.

3.10.1 Region refinement

Specific regions of further mesh refinement (lines, cones, boxes, balls, etc. within which $\max\{\vec{w}\} < h$) may also be specified by the user. The refinement process is identical to the one described for triangulated boundaries in Section 3.10, except that a specific distance function corresponding to an analytic refinement region may be defined.

For a ball refinement region of radius R and center $\vec{c}$, for example, the distance function may be implemented as:

$$d = \max\{||\vec{r} - \vec{c}|| - R, 0\} \tag{3.5}$$

Other implementations are left as an exercise to the reader.

3.11 Block clustering

It will be useful for algorithms which we will describe in the next chapter to obtain clusters of cells (exclusively within the domain) that have the same size and depth in the tree.

To do so, we must define a recursive process that finds the largest possible clusters of equally-sized cells and pushes their roots (non-leaf cells that have the clustered leaves as their children) to a list.

The recursive process is started at the tree root, and goes all the way down to its leaves. The defined function should return a tuple, which may identify:

- The maximum and minimum tree depth from the current cell to its leaves, starting from `(0, 0)` at the leaves themselves; or
- A termination signal `(-1, -1)` when the current cell contains more than one cluster, and branches of different depths.

At each cell, the minimum and maximum depth of all of its children must be obtained by recursive function calls. For leaves, this process yields, of course, `(0, 0)`.

To exclude cells that lie out of the domain, one must return the termination signal without registering new block roots whenever a leaf with index $i = 0$ is found.

For the current cell, we may define:

$$\begin{aligned} d_{min} &= \min_{c \in C} \{d_{min}^{(c)}\} \\ d_{max} &= \max_{c \in C} \{d_{max}^{(c)}\} \end{aligned} \tag{3.6}$$

If C is the set of its children.

If $d_{min} \neq d_{max}$, then the process must be terminated. All of the children of the current cell that haven't returned a termination signal when analyzed are pushed to the list, and now represent block roots. The function should, then, return `(-1,-1)`, the termination signal.

Similarly, if $d_{min} = -1 < d_{max}$, at least one of the children of the current cell has already been terminated and returned the termination signal, and the process must be terminated for the current cell as well, with non-terminated children being registered as block roots by being pushed to the list.

If $d_{min} = d_{max} \geq 0$, however, termination has not yet been reached and $(d_{min} + 1, d_{max} + 1)$ must be returned.

Finally, if a termination signal is received from all children ($d_{min} = d_{max} = -1$), it must be returned and no new block roots must be registered to the list, as lower tree levels have already been registered when the termination criterion was first hit.

The algorithm is graphically represented in Figure 3.6.

As a final addendum, the algorithm may be altered to terminate a cell if its maximum leaf depth lies beyond a certain threshold, to limit the size of the found cell clusters.

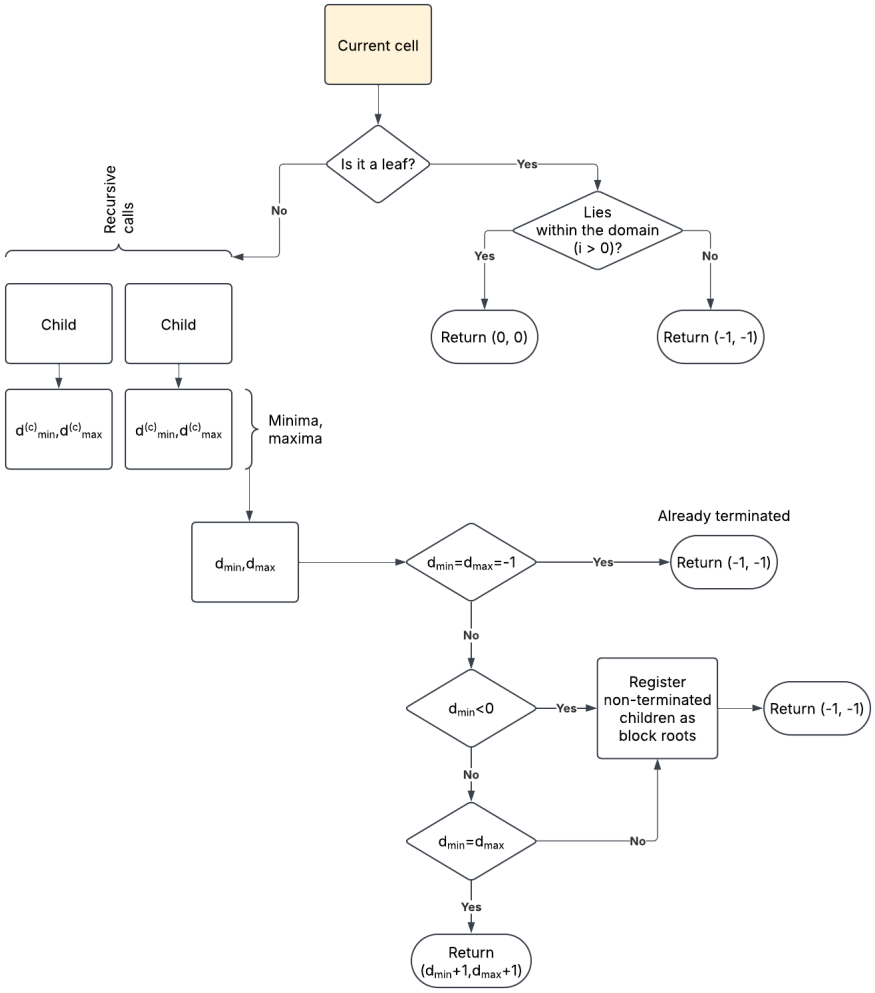

Figure 3.6: Block root detection algorithm.

3.12 Distance field and point-in-polygon optimizations

It is heavily recommended to optimize the obtention of point-in-polygon queries and distance fields using the tree data structure to one's advantage – especially in 3D mesh generation cases.

This optimization can be done trivially if the tree is traversed in a recursive manner. When a tree node corresponding to a cell of center $\vec{c}$ and widths $\vec{w}$ is reached, if $d > \frac{||\vec{w}||}{2}$ – with d indicating the distance between $\vec{c}$ and the surface –,

then its leaves will be either all within the domain, or all without it. We needn't, therefore, apply individual point-in-polygon queries to each leaf's centerpoint.

Similarly, we may approximate the projection of a set of leaves within a given upstream tree node to that of the center of said node in case $d > \frac{||\vec{w}||}{2} \times \rho$, if $\rho > 1$ is an approximation ratio.

Chapter 4

Operators and Boundary Conditions

In this chapter, we will discuss the definition of operators and boundary conditions in immersed boundary Cartesian meshes in region tree format. The algorithms described in Chapter 3 will be used extensively, and the user should review them as necessary.

4.1 Operator definition

In any Cartesian mesh, the operators for derivatives along a given direction are defined by simple finite difference or finite volume discretizations. An example is the x-axis derivative in a 2D mesh:

$$\delta u_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \tag{4.1}$$

It is vital, then, to implement the extraction of the value at adjacent cell $u_{i+1,j}$ in an efficient way.

The main particularity of the obtention of neighboring cell values is that some adjacent stencil points must be obtained "raw" from the field property vectors used to store cell values. Others, however – belonging to cells in different mesh levels –, must be obtained by interpolation from s^N neighboring cells, if s is the cell split size and N is the spatial dimensionality of the mesh. Figure 4.1 exemplifies this predicament.

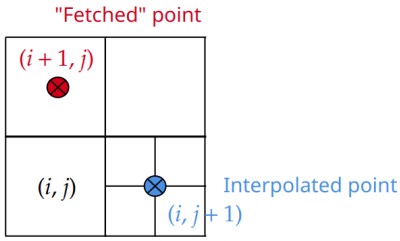

Figure 4.1: Interpolated and "fetched" stencil points due to variable region tree depths.

The implementation suggested for the obtention of these adjacent points is that of a "stencil" struct. The stencil struct should contain the information necessary to:

- Interpolate adjacent cell values from different tree levels;
- Fetch adjacent cell values from same-level neighboring cells; and
- Build a full, field variable vector by selecting between fetched and interpolated values for each stencil point.

It is recommended that such structs be stored along with the mesh and built strictly according to necessity, since they correspond to the heaviest memory usage in an IBM CFD code. An example of memory saving action is presented below for the solution of an advective equation using MUSCL reconstruction.

A generic advective equation in 2D space may be represented by:

$$\frac{\partial Q_{i,j}}{\partial t} = -\frac{\partial E_{i,j}}{\partial x} - \frac{\partial F_{i,j}}{\partial y} \tag{4.2}$$

Let us focus on the x-axis partial derivative for now. Identical considerations may be drawn for the y-axis partial derivative.

We will, using MUSCL reconstruction and the minmod slope limiter, discretize the x-axis partial derivative as:

$$\frac{\partial E_i}{\partial x} \approx \frac{E_{i+1/2} - E_{i-1/2}}{\Delta x} \tag{4.3}$$

With the flux at face $i+1/2$ being generically described by a Riemann solver expressed by function f:

$$E_{i+1/2} = f\left(Q^{(L)}_{i+1/2}, Q^{(R)}_{i+1/2}\right) \tag{4.4}$$

The values at the interface may expressed as:

$$
\begin{aligned}
Q^{(L)}_{i+1/2} &= Q_i + \frac{minmod\{Q_i - Q_{i-1}, Q_{i+1} - Q_i\}}{2} \\
Q^{(R)}_{i+1/2} &= Q_{i+1} - \frac{minmod\{Q_{i+1} - Q_i, Q_{i+2} - Q_{i+1}\}}{2}
\end{aligned}
\tag{4.5}
$$

It is evident that the calculation of flux $E_{i+1/2}$ on face $i+1/2$ would require the obtention of field property values at stencil points $i-1, i, i+1, i+2$. Two approaches may, then, be used to calculate $\frac{\partial E_{i,j}}{\partial x}$:

1. One may calculate $Q^{(R,L)}_{i-1/2,j}$ and $Q^{(R,L)}_{i+1/2,j}$ for each cell i, j, and then evaluate the Riemann solver at faces $i-1/2$ and $i+1/2$, to finally obtain $\left[f(Q^{(L)}_{i+1/2}, Q^{(R)}_{i+1/2}) - f(Q^{(L)}_{i-1/2}, Q^{(R)}_{i-1/2}\right] / \Delta x$; or

2. One may calculate $Q^{(R,L)}_{i+1/2,j}$ only, define field variable vector $\mu E_i = f(Q^{(L)}_{i+1/2}, Q^{(R)}_{i+1/2})$ and then obtain μE_{i-1} from previously defined stencil points, to then finally compute $\frac{\partial E_{i,j}}{\partial x} = (\mu E_i - \mu E_{i-1}) / \Delta x$.

Note that, as represented in Figure 4.2, approach number 2 requires the evaluation of field variables at fewer stencil points than approach number 1, which leads to significant memory savings.

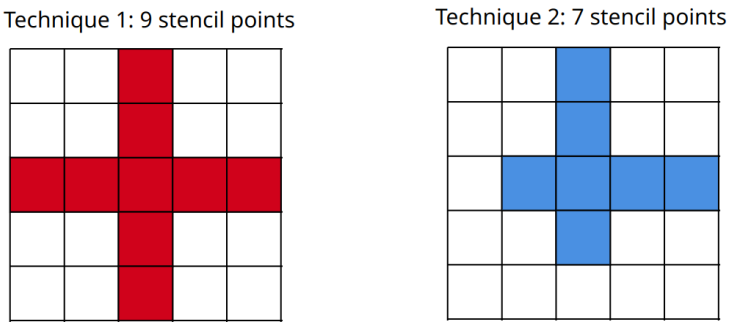

Figure 4.2: Comparison between the two divergent evaluation techniques. Fewer stencil points are required for technique 2.

Observe that the interpolations between mesh levels may be obtained as per the algorithm described in Section 3.7. Second-order accuracy via ridge regression weights can be obtained with added computational cost for pre-processing steps, only.

4.2 Boundary conditions

Boundary conditions can – for compressible CFD codes capable of accurate drag calculations – be imposed with a Ghost Cell approach. This approach is also often called the Sharp Interface Immersed Boundary Method, as it allows for discrete variations in field variables at boundaries which often occur in high-Reynolds-number flows.

Figure 4.3 shows how this method is geometrically aplied.

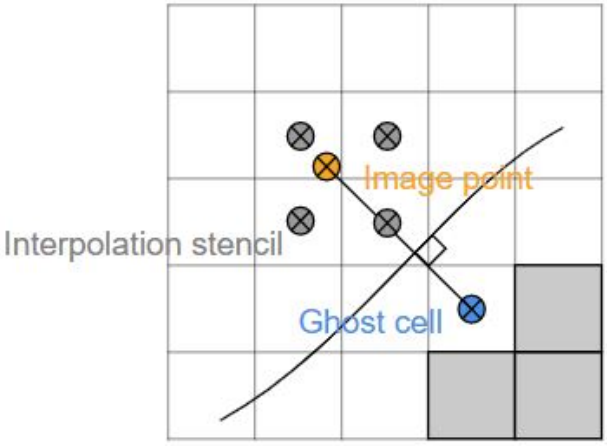

Figure 4.3: Representation of image point, ghost point, and their projections upon a boundary.

Flow variables are interpolated, from a pre-defined stencil, to an image point lying within the domain (see the interpolation techniques suggested in Section 3.7). Sherman's method (Inverse-Distance Weighing) is suggested instead of linear interpolations, as its "bounded" nature avoids that the image point values become extrapolations instead of interpolations from nearby cells, which could lead to spurious values. A formulation-agnostic boundary condition may be expressed by function g that, given the values of flow properties at image points, establishes their values at their projections to the boundary:

$$Q_B = g(Q_I) \tag{4.6}$$

We may establish an adimensional distance η between the ghost cell and the boundary so that the flow properties at a ghost cell are given by:

$$\begin{aligned} Q_G &= Q_I \times \eta + Q_B \times (1 - \eta) \\ \eta &= \frac{d_G}{d_I} \end{aligned} \tag{4.7}$$

If d_G and d_I are the signed distances between the ghost point and the boundary, and the image point and the boundary, respectively. Note that, if these are signed distances, η may be negative for ghost points lying outside of the domain.

Ghost cells may be identified as points which have an adimensional distance $\rho = d/||\vec{w}||$ from the wall lying within a given interval. When simulating a solid immersed within a fluid flow, it is preferrable to define:

$$\rho \in [-1.1; 0] \tag{4.8}$$

This guarantees that all cells which are located outside of the domain, but which may intersect the boundary, are labelled as ghost cells – and thus at least a full, watertight layer of ghost cells is tangent to any solid boundary. Also, all of the cells which are within the domain have their field property values calculated by physical equation solutions, not by mere interpolation.

Thin surfaces or hypercube boundaries, however, do not present any "internal" cells, and thus require a different interval:

$$\rho \in [0; 1.1] \tag{4.9}$$

Which is necessary to represent any surface at all, in spite of the loss of accuracy due to the absence of physical calculations for near-boundary cells.

The distance between the image point and the boundary is suggested to be defined as:

$$d_I = \max\left\{\frac{||\vec{w}||}{2} \times \sqrt{2}, d_G + \frac{||\vec{w}||}{2} \times \rho \times \sqrt{2}\right\} \tag{4.10}$$

Where N is the spatial dimensionality of the mesh.

This heuristic is chosen to guarantee that any image point will be located "at least one cell away" from any ghost cell, so that the value at the image point will not be a function of the value of the flow properties at its corresponding ghost cell – as would be the case if it was part of the image point's interpolation stencil. Also, a reasonable distance from the boundary is assured so that η does not assume excessive values which could pollute the solution in terms of numerical precision.

It is important, however, that the image point is not too far from the boundary, lest first-order numerical errors will be introduced to the boundary condition as the image point values will be too heavily influenced by cells far from the surface. If one attempts to establish a Neumann $\partial p/\partial n = 0$ boundary condition to the pressure in an Euler equations solution, the pressure at the ghost point will be defined as:

$$p_G = p_I \approx p_B + \frac{\partial p_I}{\partial n} \times d_I \tag{4.11}$$

Where $\frac{\partial p_I}{\partial n}$ is the pressure gradient at the image point, which is bound to be non-zero.

4.2.1 Imposition in explicit and implicit time-marching methods

The ghost cells must be part of the complete, field variable vectors describing solution variables. The imposition of ghost cell values – and, thus, of boundary conditions – must therefore be performed as an in-place editing of such vectors.

Their application to explicit time-marching methods is trivial: it must be done after any time step to ensure that the correct boundary conditions will be accounted for in the next round of residual computations.

With the implicit Euler method, the imposition of boundary conditions is performed by describing the implicit equation system as:

$$r = 0 \tag{4.12}$$

If residual r for equation $Q_t = f(Q)$ is given by:

$$r = \mathcal{B}\left(Q^{(n-1)} + \Delta t \times f(Q^{(n)})\right) - Q^{(n)} \tag{4.13}$$

If Δt is the time step size and $\mathcal{B}$ is an operator representing the imposition of BCs over a field variable vector.

4.3 Explicit multigrid

An explicit multigrid method can be trivially implemented by defining a cluster of cells as a group of same-size, same-depth leaf cells as per the algorithm in 3.11: it suffices to define the residual in a coarse mesh to be equal to that of the fine mesh averaged over all same-size sub-cells of each cluster.

Note, however, that coarse cells tangent to the boundary will only be correctly calculated if the residual is redefined as:

$$r = \frac{\mathcal{B}(Q^{(n-1)} + f(Q^{(n-1)}) \times \tau) - Q^{(n-1)}}{\tau} \tag{4.14}$$

If τ is a time step/pseudo-time-step length.

In other words, the averaged residuals must include boundary condition information.

4.4 Considerations on partitioning

To achieve optimum performance, it is recommended to make use of vectorization by calculating the residuals of many cells at a time in a single loop or set of vector operations. The same can be said regarding GPU kernels.

To limit memory usage and allow for better MPI parallelization, however, it would be ideal to calculate the residuals for only a batch of cells at a time, since one could then allocate merely small vectors of primitive variables which can be deallocated before moving on to the next batch.

The procedure suggested to obtain such a partitioning is:

1. To obtain the calculation stencils for each cell, among all in the domain;
2. To separate the cells among partitions using either block identification, or mere indexing. We will now refer to the cells in each partition as its image set;
3. To obtain the union of all stencils of cells in the present partition, now labelled its domain set;
4. To re-index the stencil interpolators to refer to cells in the partition's domain, only.

Figure 4.4, then, illustrates how the residuals may be calculated using the pre-defined image and domain sets along with locally-indexed stencil interpolations.

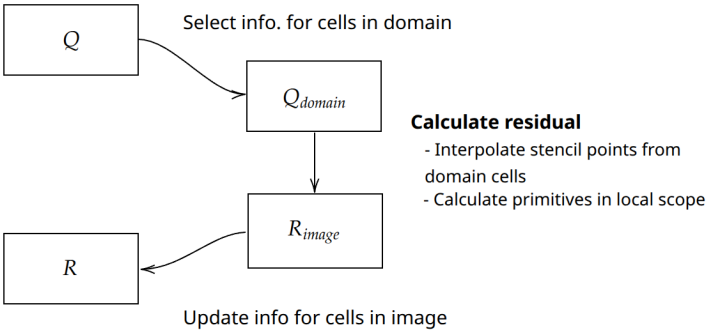

Figure 4.4: Schematic illustration regarding the calculation of residuals in a partitioned domain.

One must observe that **the stencils must include cells used to interpolate data to the image points used for boundary condition imposition**, lest the BCs will be incorrectly imposed.