PLDR Regularization for Irregular Sparse Data-Driven RANS Models

Fabiano Sasselli

Master of Science in Computational Science and Engineering

Seminar in Fluid Dynamics for CSE HS 21

Institute of Fluid Dynamics
ETH Zürich

Supervisor: Pasha Piroozmand

Professor: Prof. Dr. Patrick Jenny

Abstract

In the context of the Seminar of Fluid Dynamics for CSE, OpenFOAM code and Python scripts have been developed to enable a more flexible future testing of the discrete adjoint method with piecewise linear dimensionality reduction (PLDR) regularization proposed by researchers at the Institute of Fluid Dynamics (2021) [1] in presence of unstructured sparse reference data.

The novelty of this work consists of the the ability to automatically create a triangulation w.r.t the reference data and to artificially manufacture new reference data at specific locations such as the boundary layer or curved boundaries, and to increase the reference points resolution (i.e the number of points). Contrary to the previous framework, the new triangulation framework does not need alignment of the reference data in order to work.

This technical report aims to briefly introduce the reader to the computational method used and document the implemented framework.

Contents

1 Introduction **1**
1.1 Problem statement 1
1.2 Data assimilation as an optimization problem 2
1.3 Piecewise linear regularization 3
1.4 Optimization update 4

2 OpenFOAM and Python framework **5**
2.1 Objective 5
2.2 OpenFOAM code 5
2.2.1 Triangulation data pre-processing 5
2.2.2 Triangulation data post-processing 7
2.3 Python code 11
2.3.1 Triangle class 11
2.3.2 Nodes class 12
2.3.3 Main function 13
2.3.4 Ghost points 18
2.3.5 Boundary layer data refinement 18
2.3.6 Exemplary output 19

3 Conclusions **22**

Bibliography **23**

Attachements **24**

1 Introduction

The adjoint-based data assimilation technique has been shown to be effective in reducing model-form errors of linear eddy viscosity RANS models. A field parameter is introduced "as a correction" to the eddy viscosity. Successive gradient-based optimization of this parameter w.r.t reference data, using an adjoint method, allow to reduce RANS results errors. Regularization techniques have been applied to the field parameter to avoid unphysical solutions [3]. Recently a piece-wise linear dimensionality reduction (PLDR) technique as a regularization strategy has been developed at IFD and tested for the canonical case of flow over periodic hills using a coarsening limited to structured (Cartesian grid) reference points [1].

In this chapter, a general overview of the computational method proposed by Piroozmand *et al.* (2021) [1] is made. The main algorithm steps are shown; for detailed explanations please refer to the original manuscript.

1.1 Problem statement

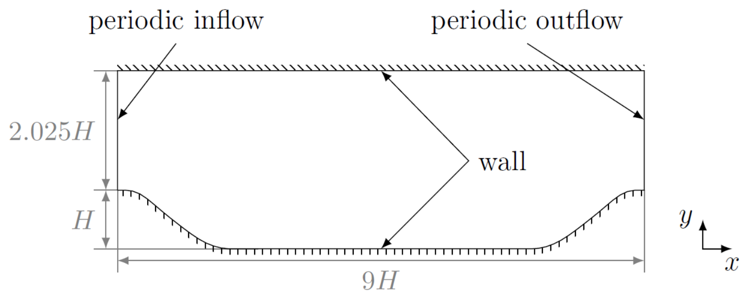

Figure 1.1: Sketch of the considered problem geometry. Shown is a periodic hill with the respective boundary conditions. Hill height H. Mean flow direction is in $x-$direction from left to right. Figure taken from Brenner *et al.* (2021) [3]

We consider the stationary, incompressible RANS equations together with the Boussinesq eddy viscosity model solved on a domain defined by a periodic hill (Fig 1.1). In the proposed model [1], it is assumed that the main source of inaccuracies is the turbulent viscosity ν_t. A correction field parameter α is multiplied to the eddy viscosity in order to correct its inherent inaccuracy. The field parameter α shall later be optimized w.r.t reference data. The resulting RANS equations reads as follows:

$$\frac{\partial \overline{u}_i}{\partial x_i} = 0 \tag{1.1}$$

$$\frac{\partial \overline{u}_i \overline{u}_j}{\partial x_j} = -\frac{\partial \overline{p}^*}{\partial x_i} + (\nu + \alpha \nu_t)\frac{\partial^2 \overline{u}_i}{\partial x_j \partial x_j} \quad ; \quad \overline{p}^* = \frac{\overline{p}}{\rho} + \frac{2}{3}k \tag{1.2}$$

$\overline{u}_j$: $j-$th component of mean flow $\overline{u}$
ν : fluid dynamic viscosity
ν_t : turbulent viscosity
α : model correction parameter
ρ : fluid density
$\overline{p}$: mean pressure field
k : turbulent kinetic energy

1.2 Data assimilation as an optimization problem

In order to optimize the correction parameter α w.r.t the given geometry and reference data, an objective/cost function is required. Brenner *et al.*, 2021, [3] proposed a bi-modal function to be used:

$$f(\alpha, \overline{\mathbf{U}}) = f_\alpha(\alpha) + f_{\overline{\mathbf{U}}}(\overline{\mathbf{U}}) \tag{1.3}$$

$\overline{\mathbf{U}} = (\overline{\mathbf{u}}, \overline{\mathbf{p}}) = (\overline{u}, \overline{v}, \overline{w}, \overline{p})$: vector of averaged state variables

The first term is utilized to correct the alpha field using a total variation (tv) and deviation (d) regularization:

$$f_\alpha(\alpha) = f_\alpha^{tv} + f_\alpha^{d}$$

with

$$f_\alpha^{tv} = C^{tv} \sum_{i \in \Omega} (\alpha_i - \alpha_i^{ini})^2 \quad ; \quad f_\alpha^{d} = C^{d} \sum_{i \in \Omega} \left[\frac{1}{|B_i|} \sum_{k \in B_i} (\alpha_i - \alpha_k)^2 \right] \tag{1.4}$$

Ω : domain
α_i : model correction parameter at cell i
α_i^{ini} : initial value of model correction parameter at cell i
C^{tv} : weighting factor of total variation
C^{d} : weighting factor of deviation from initial values
B_i : set neighbouring cells indexes
$|B_i|$: number of neighbouring cells

The second term of the cost function $f_{\overline{\mathbf{U}}}$ is used to measure discrepancies between the obtained velocity field and the reference data:

$$f_{\overline{\mathbf{U}}}(\overline{\mathbf{U}}) = \frac{\sum_j \left[\sum_{k \in \{x,y,z\}} \left[C_{j,k} (\overline{U}_{k,j} - \overline{U}_{k,j}^{ref})^2 \right] V_j \right]}{\sum_j \max_k (C_{j,k})} \tag{1.5}$$

$C_{j,k}$: indicator function for velocity component k in cell j
$\overline{U}_{k,j}$: k-th mean state variable at cell j

$\overline{U}^{ref}_{k,j}$: k-th reference mean state variable at cell j
V_j : volume of cell j

The regularization terms have been introduced by the paper authors [3] in order to penalize non-physical solutions caused by certain choices of α and to have a smooth field. Analytical formulations of the gradients of the objective function terms shall be found in the original paper [3].

Our goal is to find the optimal α field which minimizes the objective function, subject to the residuals constraint $R(\alpha, \overline{\mathbf{U}}) = 0$ and $\alpha > 0$. To this end the Lagrangian is constructed:

$$\mathcal{L}(\alpha, \overline{\mathbf{U}}) = f(\alpha, \overline{\mathbf{U}}) - \lambda^T R(\alpha, \overline{\mathbf{U}}) \tag{1.6}$$

λ : residuals sensitivity
R : residuals of discrete RANS algebraic system of equations

and the gradient w.r.t the optimization parameter α taken in order to find an expression for the derivative of the objective function w.r.t α, which will later be utilized in the Gradient Descent update rule:

$$\frac{d\mathcal{L}}{d\alpha} = \frac{\partial f_\alpha}{\partial \alpha} - \lambda^T \frac{\partial R}{\partial \alpha} = 0 \Rightarrow \frac{\partial f_\alpha}{\partial \alpha} \equiv \frac{df}{d\alpha} = \lambda^T \frac{\partial R}{\partial \alpha} \tag{1.7}$$

The residual sensitivity λ can be easily found by solving the algebraic system of equations given by

$$\left(\frac{\partial R}{\partial \overline{\mathbf{U}}}\right)^T \lambda = \left(\frac{\partial f_{\overline{\mathbf{U}}}}{\partial \overline{\mathbf{U}}}\right)^T \tag{1.8}$$

where the gradient of the residuals w.r.t the velocity field is equivalent to the system matrix of the discrete RANS equations. An analytical expression for the gradient of the residuals w.r.t alpha is found by using the alpha dependent terms in the modified RANS equation:

$$\frac{\partial R}{\partial \alpha} = -\frac{\partial}{\partial \alpha}(\nabla \cdot ((\nu_t \nabla \overline{\mathbf{u}})\alpha)) \tag{1.9}$$

$\overline{\mathbf{u}} = (\overline{u}, \overline{v}, \overline{w})$

The residual gradient w.r.t the mean pressure is ignored (not computed) beacuse there is no explicit representation of α in the Poisson equation solved for the pressure term.

1.3 Piecewise linear regularization

Piroozmand *et al.* (2021) [1] have observed that the regularizations applied to the objective function by Brenner *et al.* (2021) [3] fail to predict realistic wall shear stresses caused by an over-fitting at the wall reference points locations. They proposed an alternative approach where the objective function only retains the state variables term (Eq. 1.5):

$$f = f_{\overline{\mathbf{U}}}(\overline{\mathbf{U}}) \tag{1.10}$$

An additional regularization is then applied directly to the α field, leading to a reduction of the degree of freedom of the field. In the new approach, two dimensional piecewise linear basis functions φ_{ij} are constructed between the reference data points. The regularized α field is noted as β field.
While in the α field the values can vary between each cell without constraints, in the β field, the field values are constrained through the piecewise linear basis function by the values at the neighbouring reference data points locations (thus φ_{ij} acting as a dimensionality reduction):

$$\alpha_i = \sum_j \varphi_{ij}\beta_j \tag{1.11}$$

φ_{ij} : φ function at cell i w.r.t discrete element (triangle) j

where the gradient w.r.t β is easily found using:

$$\frac{df}{d\beta_j} = \frac{df}{d\alpha_i} \cdot \varphi_{ij} \tag{1.12}$$

As shown in the reference manuscript, this regularization is able to smooth out the alpha field and avoid over-fitting of the wall shear stress. Piroozmand *et al.* (2021) [1] showed that the obtained profiles are not satisfactory yet, maybe due to the locations of the reference points used. Different configurations of reference points locations shall be tested.

1.4 Optimization update

The optimized parameter field β (later mapped back to α) is iteratively updated using the Gradient Descent rule:

$$\beta_j^{n+1} = \beta_j^n - \delta\frac{df}{d\beta_j} \tag{1.13}$$

The specific definition of the learning rate δ and other details regarding number of iterations are found on the reference paper [1].

2 OpenFOAM and Python framework

2.1 Objective

The goal of this seminar project was to implement tools which enable a more extensive testing of the model, with a thought on future available experimental data. Ideally, the new framework should allow the creation of discrete triangles using any input set of sparse reference data points. Triangles data should then be read by the OpenFOAM code in order to compute the piecewise linear basis functions φ_{ij}. As shown in the next two sections, both requirements have been fulfilled. The need for flexible tools come from the fact that previous to this project, the OpenFOAM code utilized coding tricks to compute the triangulation. Those tricks (and hard-coded sections) would not work in presence of reference data not aligned with the domain mesh and not in Cartesian grid order. For these reasons a more general approach to the triangulation computation was needed.

2.2 OpenFOAM code

2.2.1 Triangulation data pre-processing

An OpenFOAM executable `tri_prep.C` has been implemented. This triangulation pre-processing piece of code is responsible for outputting the reference data files used by OpenFOAM (e.g MapURef) and the boundary cells coordinates in data files (`MapURef_py_pts.dat`, `BC_py_pts.dat`) which will later be read by the python scripts. This code has been useful in order to have a set of data points that could be used and modified during the development of the python scripts. The main components of the code are explained below:

- Extract the cell coordinates and export them in a file which will later be read by the Python scripts:

Listing 2.1: code extract from tri_prep.C

```
//EXPORT IN MapURef_py_pts.dat THE REFERENCE PTS COORDINATES
//USED FOR FOR TRIANGULATION ROUTINE IN PYTHON
Info<< "Exporting reference data to MapURef_py_pts.dat\n" << endl;
std::ofstream py_MapURef_file_obj;
py_MapURef_file_obj.open("MapURef_py_pts.dat", std::ios::out
                         | std::ios::app);
forAll(MapURefFine, celli)
{
        if(MapURefFine[celli] > 0.0)
        {

                double x = (double)mesh.C()[celli][0];
                double y = (double)mesh.C()[celli][1];
                double z = (double)mesh.C()[celli][2];

                //print in the .dat file the coordinates
                py_MapURef_file_obj << x << "; " << y << "; " << z
                                        << std::endl;
        }
}
py_MapURef_file_obj.close();
```

- From the meshed geometry, extract the boundary faces coordinates and export them in a file which will later be read by the Python scripts:

Listing 2.2: code extract from tri_prep.C

```
//EXPORT IN BC_py_pts.dat THE BOUNDARIES COORDINATES
//USED FOR FOR TRIANGULATION ROUTINE IN PYTHON

Info<< "Exporting reference data to BC_py_pts.dat\n" << endl;

std::ofstream py_BC_file_obj;
py_BC_file_obj.open("BC_py_pts.dat", std::ios::out | std::ios::app);
forAll(mesh.C().boundaryField(), patchI)
{
        const fvPatchVectorField& mf = mesh.C().boundaryField()[patchI];

        forAll(mf, faceI)
        {
        double tmp_x =mf[faceI][0];
        double tmp_y =mf[faceI][1];
        double tmp_z =mf[faceI][2];
        py_BC_file_obj << tmp_x << "; " << tmp_y << "; " << tmp_z
                                << std::endl;
        }
}
py_BC_file_obj.close();
```

2.2.2 Triangulation data post-processing

The existing OpenFOAM executable `AdjointBasisFoam.C` and some of its header files have been modified in order to read the python script output data and compute the basis functions accordingly. The main modifications have been applied to `basis.H`. Minor modifications have also been applied to other header files. Extracts from the code present in `basis.H` are shown below:

- Read the Python script output file `unique_nodes.dat`. This file contains a list of coordinates of the unique, i.e no duplicates, nodes of the triangulation. First the file is opened in order to get the number of rows of the file:

Listing 2.3: code extract from basis.H

```
//RETRIEVE FROM PYTHON THE NUMBER OF UNIQUE NODES
//necessary since the tri_nodes.dat contains the list of triangles indeces
//and the nodes coordinates they are made of

Info<< "Retrieve␣Python␣data\n" << endl;
int number_of_unique_nodes = 0;
std::ifstream node_file_obj("unique_nodes.dat");
if(node_file_obj.is_open())
{

        std::string line;
        while (std::getline(node_file_obj, line))
        {
                ++number_of_unique_nodes;
        }
}

node_file_obj.close();
```

Afterwards the file is opened to retrieve the data:

Listing 2.4: code extract from basis.H

```

//READ FROM unique_nodes.dat"THE NODES VALUES AND STORE
//THEM INTO THE ref_pts array of structures

Cmpnts *ref_pts;
ref_pts = (Cmpnts*) malloc(number_of_unique_nodes * sizeof (Cmpnts));

#include <string>
std::ifstream node_file_obj_2;
node_file_obj_2.open("unique_nodes.dat");
if(node_file_obj_2.is_open())
{
        double x, y, z;
        for(int n = 0; n<number_of_unique_nodes ; ++n)
        {
            node_file_obj_2 >> x >> y;

            ref_pts[n].x = x;
            ref_pts[n].y = y;
            ref_pts[n].z = 0.0;

            if (node_file_obj_2.eof())
```

```
                {
                    break;
                }
            }
}
node_file_obj_2.close();
```

- Read the Python script output file `tri_nodes.dat`. This file contains a list of triangle nodes. In each row, indexes of three nodes (i.e of a single triangle) are stored. These indexes correspond to the line number in the `unique_nodes.dat` file. First the file is opened in order to get the number of rows of the file:

Listing 2.5: code extract from basis.H

```
//OPEN ONE TIME tri_nodes.dat and unique_nodes TO OBTAIN ITS LENGTH
//WHICH IS USED LATER TO INITIALIZE AN ARRAY OF STRUCTS

int number_of_tri = 0;
std::ifstream tri_file_obj("tri_nodes.dat");
if(tri_file_obj.is_open())
{
        std::string line;
        while (std::getline(tri_file_obj, line))
        {
                ++number_of_tri;
        }
}
tri_file_obj.close();
```

Afterwards the file is opened for retrieving the data. The data is stored in an array of structures. Where each struct represent a single triangle and contains the nodes indexes:

Listing 2.6: code extract from basis.H

```
//CREATE TRIANGLE DATA STRUCTURE WHICH CONTAINS TRIANGLE
//NODES INDECES AND TRIANGLE NUMBER
Info<< "Create triangles\n" << endl;
Cmpnts *triangle;
triangle = (Cmpnts*) malloc(number_of_tri * sizeof (Cmpnts));
std::ifstream tri_file_obj_2;
tri_file_obj_2.open("tri_nodes.dat");

if(tri_file_obj_2.is_open())
{
        double n_A, n_B, n_C;
        for(int t = 0; t<number_of_tri ; ++t)
        {
                tri_file_obj_2 >> n_A >> n_B >> n_C;

                triangle[t].x = n_A;     //node A
                triangle[t].y = n_B;     //node B
                triangle[t].z = n_C;     //node C
                triangle[t].index = t;

                if (node_file_obj_2.eof())
                {
                        break;
                }
        }
}
```

```
tri_file_obj_2.close();
```

- Create a OpenFOAM object which stores data for each mesh cell. The data stored for each cell correspond to the triangle index/number that the mesh cell is part of:

Listing 2.7: code extract from basis.H

```
//OPEN FOAM MESH BASED OBJECT WHICH ASSIGN TO EACH MESH CELL
//A TRIANGLE NUMBER THAT CELL IS PART OF

volScalarField whichTriangle
(
    IOobject
    (
        "whichTriangle",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("whichTriangle", dimless, -1.0)
);


Cmpnts A, B, C;
forAll(whichTriangle, celli)
{
        for (int i=0; i< number_of_tri; ++i)
        {
                //retrieve the point number of the triangle nodes
                //numbering based on ref_pts
                int nodeAIndex = triangle[i].x;
                int nodeBIndex = triangle[i].y;
                int nodeCIndex = triangle[i].z;

                //retrieve the nodes A,B,C
                A = ref_pts[nodeAIndex];
                B = ref_pts[nodeBIndex];
                C = ref_pts[nodeCIndex];


                //check if our mesh point is inside the triangle ABC
                // https://stackoverflow.com/questions/2049582/how-to-determine-if
                // -a-point-is-in-a-2d-triangle
                float d1, d2, d3;
                bool has_neg, has_pos, res;

                d1 = (mesh.C()[celli][0] - B.x) * (A.y - B.y)
                                - (A.x - B.x) * (mesh.C()[celli][1] - B.y);
                d2 = (mesh.C()[celli][0] - C.x) * (B.y - C.y)
                                - (B.x - C.x) * (mesh.C()[celli][1] - C.y);
                d3 = (mesh.C()[celli][0] - A.x) * (C.y - A.y)
                                - (C.x - A.x) * (mesh.C()[celli][1] - A.y);

                has_neg = (d1 < 0) || (d2 < 0) || (d3 < 0);
                has_pos = (d1 > 0) || (d2 > 0) || (d3 > 0);
                res = !(has_neg && has_pos);

                if(res)
                {
                        whichTriangle[celli] = triangle[i].index;
                        break;

                }
        }
}
```

- Computing the piece-wise linear basis function φ_{ij} for each mesh cell:

Listing 2.8: code extract from basis.H

```
// computes Phi only once for the whole opt procedure
forAll(Phi, celli)
{
        int triangle_num = whichTriangle[celli];
        int nodeAIndex = triangle[triangle_num].x;
        int nodeBIndex = triangle[triangle_num].y;
        int nodeCIndex = triangle[triangle_num].z;

        double denominator = (
                                (ref_pts[nodeAIndex].x-ref_pts[nodeCIndex].x)
                           *(ref_pts[nodeBIndex].y-ref_pts[nodeCIndex].y)
                           -(ref_pts[nodeBIndex].x-ref_pts[nodeCIndex].x)
                       *(ref_pts[nodeAIndex].y-ref_pts[nodeCIndex].y)
                         );

        Phi[celli][0] = (
                                (ref_pts[nodeBIndex].y-ref_pts[nodeCIndex].y)
                           * mesh.C()[celli][0]
                           +(ref_pts[nodeCIndex].x-ref_pts[nodeBIndex].x)
               * mesh.C()[celli][1]
                           - ref_pts[nodeCIndex].x*ref_pts[nodeBIndex].y
                           + ref_pts[nodeBIndex].x*ref_pts[nodeCIndex].y
                         ) / denominator;

        Phi[celli][1] = (
                                (ref_pts[nodeCIndex].y-ref_pts[nodeAIndex].y)
                           * mesh.C()[celli][0]
                           +(ref_pts[nodeAIndex].x-ref_pts[nodeCIndex].x)
               * mesh.C()[celli][1]
                           + ref_pts[nodeCIndex].x*ref_pts[nodeAIndex].y
                           - ref_pts[nodeAIndex].x*ref_pts[nodeCIndex].y
                         ) / denominator;

        Phi[celli][2] = (
                                (ref_pts[nodeAIndex].y-ref_pts[nodeBIndex].y)
                           * mesh.C()[celli][0]
                           +(ref_pts[nodeBIndex].x-ref_pts[nodeAIndex].x)
               * mesh.C()[celli][1]
                           + ref_pts[nodeAIndex].x*ref_pts[nodeBIndex].y
                           - ref_pts[nodeBIndex].x*ref_pts[nodeAIndex].y
                         ) / denominator;
}
```

2.3 Python code

The triangulation script has been written in Python. Two classes have been defined: one is the nodes class and contains methods to modify or add reference points, the second one is the triangle class, which is responsible for creating the triangulation using the `scipy.spatial.Delunay` function. The Delunay triangulation algorithm ensures the creation of triangles while minimizing the number of extremely skewed/acute angles. By definition the Delunay triangulation is a triangulation such that for a set of points, no vertex lies inside the circumcircles of any other triangle [4]. The Delunay triangulation is common in meshing software and was already efficiently implemented in the python SciPy library, thus the choice of using it. A general sketch of the Delunay triangulation applied to a set of points can be seen in Fig 2.1.

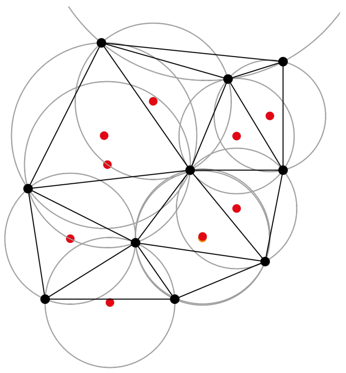

Figure 2.1: Delunay triangulation of a set of points (black points). Red points represent the center points of the triangles circumcircles [4].

In the main function of the code, the user has to correctly utilize the provided class methods in order to modify (e.g add) the existing reference points; an example is presented in the attached code. Depending on the geometry (e.g enclosed vs open geometry) certain parameters might have to be tuned accordingly. Comments indicate when this modifications are necessary. The python script is automatically called by the optimizer at run time.

2.3.1 Triangle class

The triangle class is used to create a triangulation and return data related to the triangles. Below the implementation of the class:

Listing 2.9: code extract from triangulation.py

```
#TRIANGLES CLASS

class Triangle:

    #this class is responsible for creating the triangle object
    def __init__(self, ref_data_pts):
        self.nodes = ref_data_pts

    def create_unstructred_mesh(self):
        triangles = Delaunay(self.nodes)

        return triangles#scipy.spatial methods can be applied to this object

    def get_triangles_nodes_list(self, triangles):
        nodes_list = triangles.simplices

        return nodes_list
```

2.3.2 Nodes class

The nodes class contains methods to modify and add reference data points. Below a brief explanation of each class method. Please refer to the original code to see the class implementation. The nodes class methods allow to:

- `extract_i_component`: extract the desired coordinate from a list of points ($i = 1 \rightarrow$ extract x component, $i = 2 \rightarrow$ extract y component)
- `extract_all_components`: extract both coordinates data from a list of points and returns two lists.
- `merge_xy_coordinates`: from two list of individual coordinates, create a single list of points
- `compute_mesh_size_dimensions`: compute the span in x,y, direction of the mesh
- `scalar_multiply_pts`: multiply points coordinates by a scalar value
- `get_centroid_shift`: compute centroid of a list of points
- `translate_points`:shift in two directions by a scalar value the points
- `untranslate_points`: translate back to the original position points who had been previously translated
- `get_pts_list_to_dict`:
- `get_triangle_centers`: given a list of triangles, returns a list containing the triangles centers coordinates.
- `get_points_distance_statistics`: compute the average distance between points. OpenFOAM exports the boundary points not in order. When applying other methods which utilize mean positions w.r.t points, we may obtain points in undesired locations. This method prevent that by returning the average points distance, which can be used to ensure points are not placed too far away.
- `get_curvature_points`:iterate across the list of boundary points and check the inclination angle between segments of points. If not null, then the points part of the segment are considered as on a curved surface.

- `add_points_to_curvature`: given a list of points on curved boundaries, add points to these curved boundaries
- `stretched_boundary_layer`:given a list which contains boundary points and their respective furthest boundary layer points; create new points using the stretched mesh approach between the two end points.
- `get_equispaced_points`: given a list of points, get a sliced list containing one every n points of the original list.
- `get_closest_bc_pts`: return the boundary points which are closest to existing reference points
- `create_boundary_layer_edge_pts`: given a list of boundary points, shifts those points in order to construct the other end of the boundary layer.
- `get_y_minus_boundary_layer_pts`: given a list of points pairs, return the points pairs which are on the lower part of the geoemtry (works well for the periodic hill).

2.3.3 Main function

In the main function, modifications shall be made between the node class initialization and the node list output. Additional modifications can be made after the triangle class has been initialized. Shown below is an example on how to merge the different methods together in order to obtain the desired output.

- Function to read the OpenFOAM exported data, defined outside of the main function

Listing 2.10: code extract from triangulation.py

```
def read_data(path):
    #extract data from format x;y;z
    file_obj = open(path, "r")
    lines_lst = [line.rstrip() for line in file_obj]
    lines_lst = [line.split(";") for line in lines_lst]
    lines_lst = [np.float_(line)  for line in lines_lst]
    lines_lst = [line.tolist()  for line in lines_lst]
    #outputs a list containing sublists with x,y,z coordinates
    #[[x,y,z],[x,y,z],...]

    return lines_lst
```

- Definition of settings uzilized for the triangulation and for the nodes class methods.

Listing 2.11: code extract from triangulation.py

```
#DEFINE GENERAL PARAMETERS/SETTINGS
    use_reference_for_ghost_points = False
    use_reference_for_boundary_layer = True
    #if true it will utilize reference points closest to boundary to create
    #ghost pts
    #if false will utilize a set of equidsitant points of the boundary
    #to create ghost pts
    include_node_center = False
    include_node_boundary_layer = True
    include_node_high_gradient = True
    make_plot = True
    #Boundary Layer settings
    #   number of points to generate at the boundary layer
    N_BL = 15
    d_BL = 0.2
    #   stretching parameter, in the slides should be positive,
    #but here works only if negative,
    #probably due to fact that we are in negative part of domain
    b = −30

    scale_multiplier = 0.2
```

- Reading input data

Listing 2.12: code extract from triangulation.py

```
#READ OpenFOAM EXPORTED DATA

    #path settings, do not change
    BC_path = 'BC_py_pts.dat'
    ref_path = 'MapURef_py_pts.dat'
    out_tri_path = 'tri_nodes.dat'
    out_ref_path = 'unique_nodes.dat'

    ref_data_pts = read_data(ref_path)
    BC_data_pts = read_data(BC_path)
```

- Initialize the nodes object, compute the centroid and center the data. Centering at (0,0) the data is important for the correct functioning of class methods.

Listing 2.13: code extract from triangulation.py

```
 #INITIALIZE NODES CLASS

    nodes_obj = Nodes(ref_data_pts, BC_data_pts, N_BL, d_BL, b, scale_multiplier)

    #shift the data s.t origin is the center
    shift_x, shift_y = nodes_obj.get_centroid_shift(nodes_obj.bc_data)
    nodes_obj.bc_data = nodes_obj.translate_points(nodes_obj.bc_data, shift_x
                                , shift_y)
    nodes_obj.ref_data = nodes_obj.translate_points(nodes_obj.ref_data, shift_x
                                , shift_y)

    #get boudnary points distance statistics
    mean_dist, std_dev_dist =
                         nodes_obj.get_points_distance_statistics(nodes_obj.bc_data)

    #get mesh size
    size_x = nodes_obj.compute_mesh_size_dimension_i(nodes_obj.bc_data,1)
    size_y = nodes_obj.compute_mesh_size_dimension_i(nodes_obj.bc_data, 2)

    #get closest reference points to the boundary
    bc_closest_nodes =
                 nodes_obj.get_closest_bc_pts(nodes_obj.bc_data, nodes_obj.ref_data)
```

- Using the Cartesian grid of reference points, creates an inner boundary layer on the bottom wall. Afterwards the original list of reference points is overwritten with the newly generated ones.

Listing 2.14: code extract from triangulation.py

```
# CREATE BOUNDARY LAYER REFERENCE POINTS
    if include_node_boundary_layer == True:
        if use_reference_for_boundary_layer == True:
            boundary_layer_list =
                  nodes_obj.get_y_minus_boundary_layer_pts(bc_closest_nodes)
            boundary_layer_nodes =
                  nodes_obj.stretched_boundary_layer(boundary_layer_list)
            nodes_obj.ref_data = boundary_layer_nodes
            #nodes_obj.ref_data += boundary_layer_nodes
        else:
            bl_dict = nodes_obj.create_boundary_layer_edge_pts(nodes_obj.bc_data)
            boundary_layer_list = nodes_obj.get_y_minus_boundary_layer_pts(bl_dict)
            boundary_layer_nodes =
                        nodes_obj.stretched_boundary_layer(boundary_layer_list)
            nodes_obj.ref_data += boundary_layer_nodes
```

- Add reference points at curved boundary surfaces

Listing 2.15: code extract from triangulation.py

```
#HIGH GRADIENT NODE REFINEMENT
    if include_node_high_gradient == True:
        nodes_on_curve =
             nodes_obj.get_curvature_points(nodes_obj.bc_data, mean_dist
                                                   , std_dev_dist)
        curve_mid_nodes = nodes_obj.add_points_to_curvature(nodes_on_curve)
        nodes_obj.ref_data+=curve_mid_nodes
```

- Create ghost points w.r.t equispaced boundary points or w.r.t existing reference points. In this example, since we only have reference points at the bottom boundary layer, we have to generate ghost points w.r.t equispaced boundary points.

Listing 2.16: code extract from triangulation.py

```
# CREATE GHOST REFERENCE POINTS
    #always last step before triangulation
    if use_reference_for_ghost_points == True:
        ghost_points = nodes_obj.scalar_multiply_pts(bc_closest_nodes, size_x
                                      , size_y,scale_multiplier)
    else:
        n_space = 10
        equspaced_bc_pts =
                 nodes_obj.get_equispaced_points(n_space, nodes_obj.bc_data)
        equi_pts_dict = {}
        for i in range(len(equspaced_bc_pts)):
            equi_pts_dict.update({tuple(equspaced_bc_pts[i]): []})
        ghost_points = nodes_obj.scalar_multiply_pts(equi_pts_dict, size_x, size_y,
                                     scale_multiplier)
    nodes_obj.ref_data+=ghost_points
```

- Output the list of unique reference points (ghost points included)

Listing 2.17: code extract from triangulation.py

```
# OUTPUT LIST OF THE NODES

    mod_ref_data = nodes_obj.ref_data

    #output a data file containing the updated reference points
    # (old reference + ghost)
    #position of the points in the list will correspond to the node index
    #node_0_x node_0_y
    #node_1_x node_1_y
    #node_2_x node_2_y
    #...
    file_obj = open(out_ref_path, "w")
    file_obj.close()
    file_obj = open(out_ref_path, "a")

    nodes = nodes_obj.untranslate_points(nodes_obj.ref_data, shift_x, shift_y)
    nodes = np.array(nodes)
    for k in range(len(nodes)):
        #shift nodes back to pre-shift position

        #format floating point representation and transform it into string
        x_n = nodes[k][0]
        y_n = nodes[k][1]
        x_n = str("{0:.6f}".format(x_n))
        y_n = str("{0:.6f}".format(y_n))
        str_to_print = x_n+" "+y_n+ "\n"
        file_obj.write(str_to_print)
    file_obj.close()
```

- Compute triangulation

Listing 2.18: code extract from triangulation.py

```
#INITIALIZE TRIANGLE CLASS

    tri_obj = Triangle(mod_ref_data)
    tri = tri_obj.create_unstructred_mesh()
    tri_vertex_lst = tri_obj.get_triangles_nodes_list(tri)
```

- If needed, get triangles center points and redo triangulation on an update list of reference points.

Listing 2.19: code extract from triangulation.py

```
# COMPUTE TRIANGLES CENTERS
    if include_node_center == True:
        nodes_centers = nodes_obj.get_triangle_centers(nodes_obj.ref_data,
                                tri_vertex_lst)
        nodes_obj.ref_data+=nodes_centers
        mod_ref_data = nodes_obj.ref_data

        tri_obj = Triangle(mod_ref_data)
        tri = tri_obj.create_unstructred_mesh()
        tri_vertex_lst = tri_obj.get_triangles_nodes_list(tri)
```

- Output the list of triangles nodes indexes.

Listing 2.20: code extract from triangulation.py

```
# CHECK WHICH NODES ARE PART OF WHICH TRIANGLE
    # AND PRINT IT IN A FILE
    # output format: node_a node_b node_c
    # each line correspond to a different triangle, implicit indexing

    file_obj = open(out_tri_path, "w")
    file_obj.close()
    file_obj = open(out_tri_path, "a")

    for k in range(len(tri_vertex_lst)):
        tri_k_vertex_a = tri_vertex_lst[k][0]
        tri_k_vertex_b = tri_vertex_lst[k][1]
        tri_k_vertex_c = tri_vertex_lst[k][2]
        # nodes are shifted back to their true position
        # since intial data had been shifted
        tri_nodes = str(tri_k_vertex_a) + " " + str(tri_k_vertex_b)
        + " " + str(tri_k_vertex_c) + "\n"
        file_obj.write(tri_nodes)

    file_obj.close()
```

- Plot the desired points and triangles

Listing 2.21: code extract from triangulation.py

```
#PLOTTING
    if make_plot == True:
        x_bc, y_bc = nodes_obj.extract_all_components(nodes_obj.bc_data)
        x_ref, y_ref = nodes_obj.extract_all_components(nodes_obj.ref_data)
        x_gh, y_gh =  nodes_obj.extract_all_components(ghost_points)
        fig, ax = plt.subplots(figsize=(15, 7))
        plt.title("Triangulation")
        ax.set_xlabel('$x [-]$')
        ax.set_ylabel('$y [-]$')
        plt.triplot(x_ref, y_ref, tri_vertex_lst,  color='black', linewidth=0.3)
        plt.scatter(x_bc, y_bc, color='red',
                    linewidth=1, s=3, label='boundary points')
        plt.scatter(x_ref, y_ref, color='k', linewidth=1, s = 5,
                    label = 'reference points')
        plt.scatter(x_gh, y_gh, color='orange',
                    linewidth=1, s=20, label='ghost points')
        plt.legend(loc="best")
        plt.tight_layout()
        plt.savefig("tri_ref_pts.png")
        plt.show()

    print("--- %s seconds ---" % (time.time() - start_time))
```

The shown code first creates additional boundary layer reference points and overwrite the original set of Cartesian grid reference points with the newly created boundary layer points. To avoid the points being overwritten, the user simply has to comment and uncomment the last two lines of the if clause in the boundary layer code block. Afterwards additional reference points are added at the curves of the periodic hill. Finally ghost points are inserted w.r.t the boundary points (and not the reference points). A plot of the produced reference points is found in Fig 2.2.

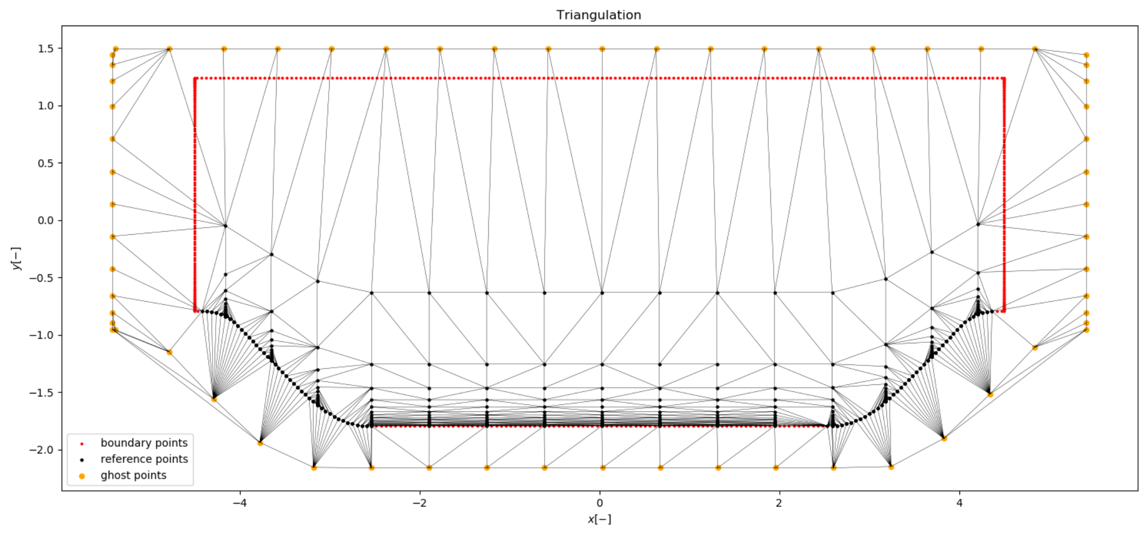

Figure 2.2: Image of the considered periodic hill points geometry. Shown in red are the boundary points, in yellow the ghost points (away from the main geometry) and in black the reference data points. Lines between points define the triangles boundaries. This configuration of reference points has been created using the code shown in this chapter.

2.3.4 Ghost points

Boundary cells shall be considered for the computation of β field too. This is not possible by default if reference points are not present at the boundary, i.e if the boundary cells are not included in any discrete triangle. To this end it is necessary to create ghost points outside of the boundary to enable the inclusion of boundary cells in the triangles.
The created ghost points have to be located in the far field of the domain to ensure that the computed basis function φ_{ij}, at the boundary cells, has the same value as the closest reference data point. The Python script offers the possibility to create ghost points using existing reference points (useful only if they are located close to all the boundaries) or by using equidistant boundary points.

2.3.5 Boundary layer data refinement

At wall boundaries the spatial gradients of pressure and velocity have a high degree of variability, especially in turbulent flows, therefore in CFD a coarser grid is usually required to resolve correctly the flow in such regions. This feature has to be present in our framework too in order to recover a physically accurate flow field. Methods in the nodes class allow the user to insert additional reference points at the bottom wall boundary layer. The points are inserted with an exponentially decaying distance when moving towards the wall. This has been implemented with a stretched mesh algorithm commonly utilized for meshing. The stretched grid is obtained by considering the coordinates mapping

$$\mathcal{R}^2 \to \mathcal{R}^2 \quad : \qquad (\xi, \eta) \mapsto (x, y) = \left(x, h\frac{(\beta+1) - (\beta-1)B^{1-\eta}}{1 + B^{1-\eta}} \right) \tag{2.1}$$

with

$$B = \frac{\beta+1}{\beta-1} \quad ; \quad \beta : \text{stretching parameter } 1 < \beta < \infty$$

h is the height of our stretched domain of interest. The equations have been taken by MIT, 2015 [2]

2.3.6 Exemplary output

The following figures show possible outputs that can be generated using the scripts. A Cartesian grid of reference points is used as input data for the first three examples. The grid is successively modified by introducing a curvature refinement (Fig 2.3), boundary layer (Fig 2.4), and triangles center nodes (Fig 2.5). Figure 2.6 and 2.7 show two cases where the input data consist of random unstructured reference points. An additional curvature refinement is applied to both cases and boundary layer points created for one of them (Fig 2.7).

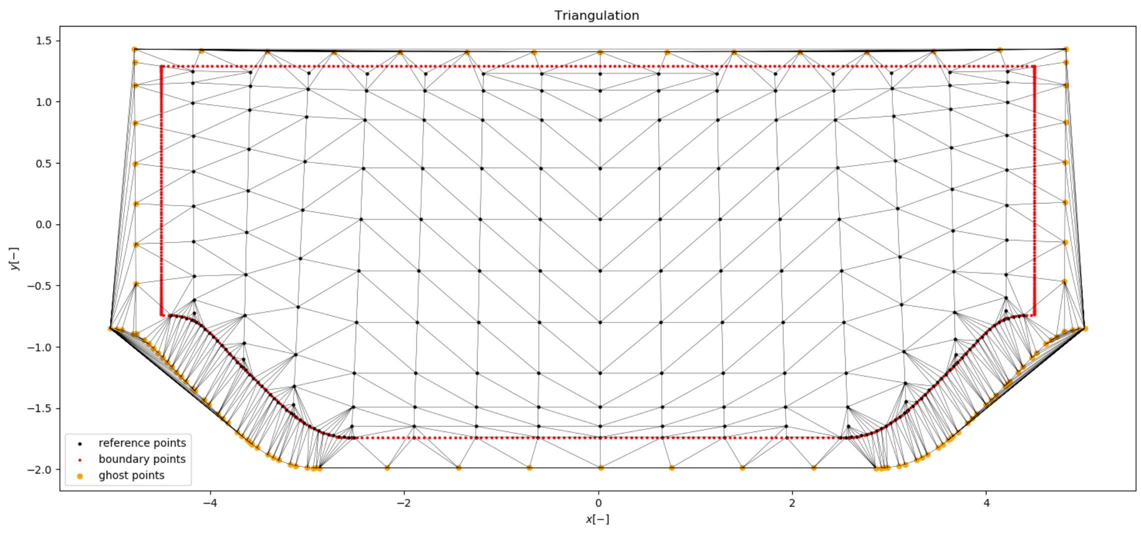

Figure 2.3: Image of the considered periodic hill points geometry. Shown in red are the boundary points, in yellow the ghost points (away from the main geometry) and in black the reference data points. Lines between points define the triangles boundaries. This case has been created starting from a structured Cartesian grid of reference points. New reference points have been added at the curved boundaries. Ghost points added using existing reference points for their creation.

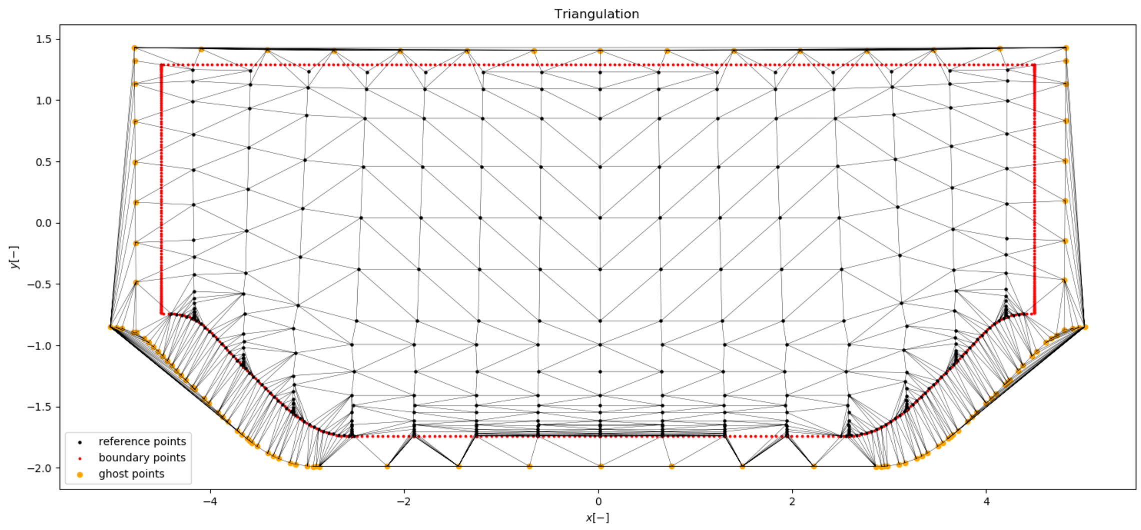

Figure 2.4: Image of the considered periodic hill points geometry. Shown in red are the boundary points, in yellow the ghost points (away from the main geometry) and in black the reference data points. Lines between points define the triangles boundaries. This case has been created starting from a structured Cartesian grid of reference points. New reference points have been added at the curved boundaries and at the bottom wall boundary layer. Ghost points added using existing reference points for their creation.

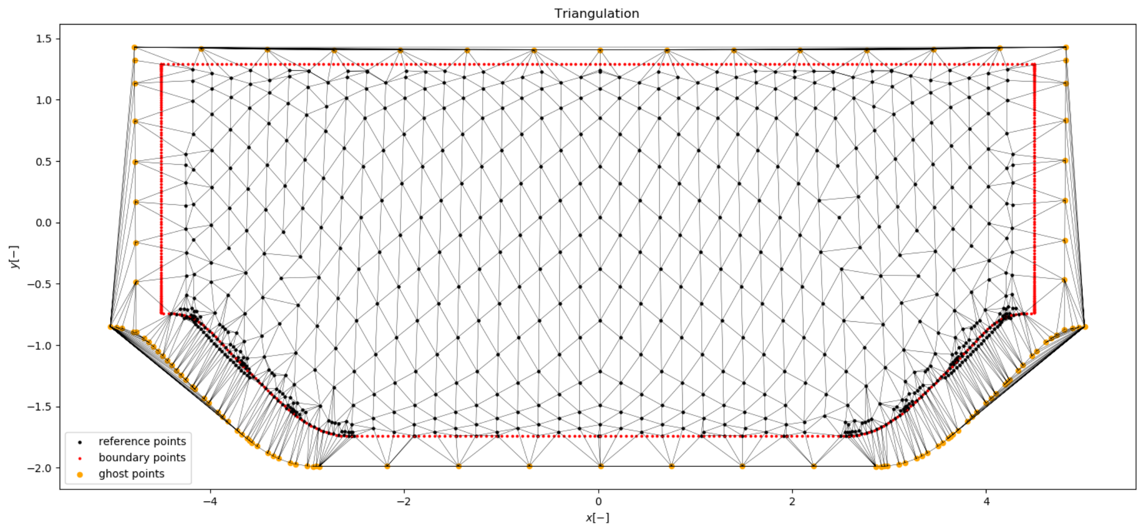

Figure 2.5: Image of the considered periodic hill points geometry. Shown in red are the boundary points, in yellow the ghost points (away from the main geometry) and in black the reference data points. Lines between points define the triangles boundaries. This case has been created starting from a structured Cartesian grid of reference points. New reference points have been added at the curved boundaries. Afterwards a triangulation has been made, and new reference points added at the center of the create triangles. Finally a second triangulation has been made using the modified data set.

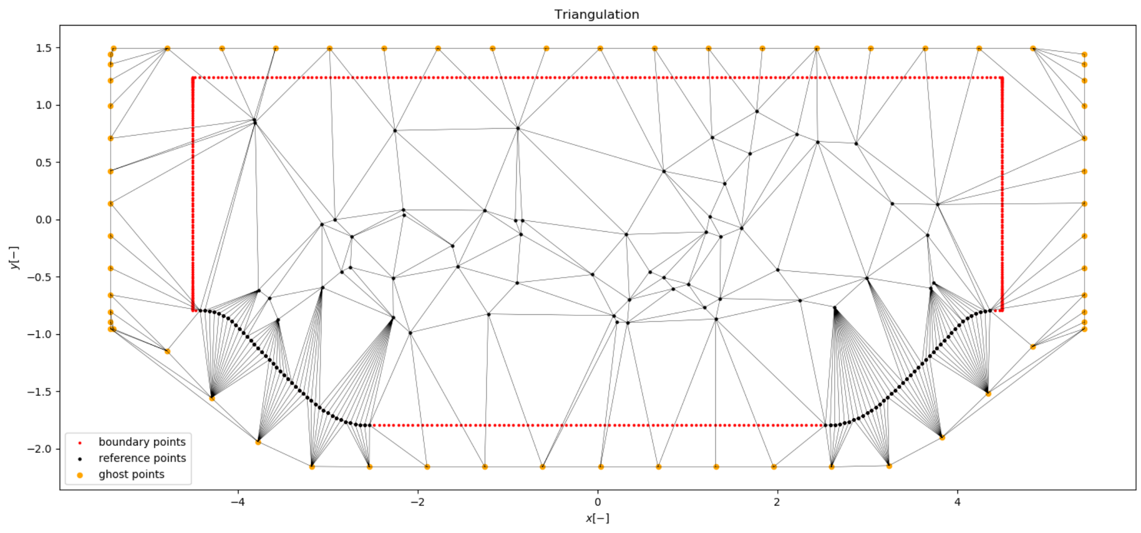

Figure 2.6: Image of the considered periodic hill points geometry. Shown in red are the boundary points, in yellow the ghost points (away from the main geometry) and in black the reference data points. Lines between points define the triangles boundaries. This case has been created starting from a set of uniformly distributed random points. New reference points have been added at the curved boundaries. Ghost points added using boundary points for their creation.

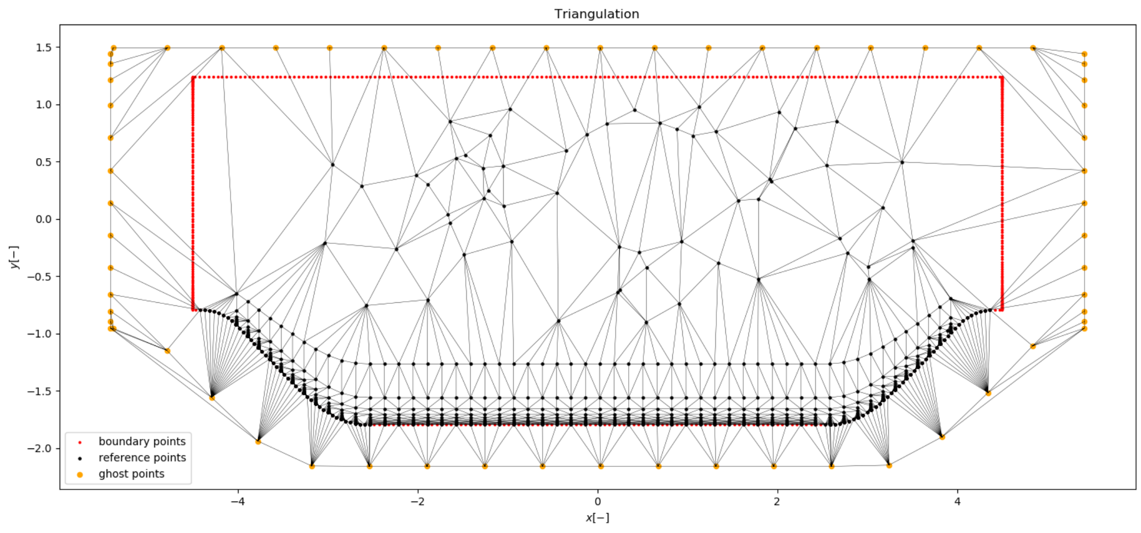

Figure 2.7: Image of the considered periodic hill points geometry. Shown in red are the boundary points, in yellow the ghost points (away from the main geometry) and in black the reference data points. Lines between points define the triangles boundaries. This case has been created starting from a set of uniformly distributed random points. New reference points have been added at the curved boundaries and at the bottom wall boundary layer. Ghost points added using boundary points for their creation.

3 Conclusions

In this work, tools to enable a more in-depth and flexible testing of the Piecewise Linear Regularization proposed by Piroozmand *et al.* (2021) [1] have been created. This additional regularization has been introduced by the paper authors to better reconstruct the wall shear stresses at the walls. The possibility to construct a discrete triangulation based on given reference data has been implemented, together with the possibility to modify existing reference data and feed all the data automatically to the OpenFOAM code.
Further extensions to this framework should include a more general approach to generate ghost points and boundary layer points in the case of complex, highly curved geometries. In addition, imposing Neumann boundary condition for the beta field at the boundaries need a further treatment in the case of very non-convex boundaries. Also, the framework currently works only on 2-dimensional geometries and the 3-dimensional triangulation is another possible future work. Such improvements increase the generalizability of the approach to more complex geometries.

Bibliography

[1] Pasha Piroozmand *et al.*2021.DIMENSION REDUCTION FOR SPARSE DATA DRIVEN RANS MODELS.Preprint submitted to Journal of Computational Physics, Institute of Fluid Dynamics ETH Zurich.

[2] Prof. Pierre Lermusiaux.2015.NUMERICAL FLUID MECHANICS.Lecture notes, lecture 21-slide 27, Massachusetts Institute of Technology.`https://ocw.mit.edu/courses/mechanical-engineering/2-29-numerical-fluid-mechanics-spring-2015/lecture-notes-and-references/MIT2_29S15_Lecture21.pdf`

[3] Oliver Brenner *et al.*2021.EFFICIENT ASSIMILATION OF SPARSE DATA INTO RANS-BASED TURBULENT FLOW SIMULATIONS USING A DISCRETE ADJOINT METHOD.Preprint submitted to Journal of Computational Physics, Institute of Fluid Dynamics ETH Zurich.

[4] Wikipedia the Free Encyclopedia.2021.DELUNAY TRIANGULATION. `https://en.wikipedia.org/wiki/Delaunay_triangulation`

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PLDR Regularization for Irregular Sparse Data-Driven RANS Models

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
FABIANO	SASSELLI

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
17.12.2021, Zurich	Fabiano Sasselli

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.