Chapter 1

Introduction

The aim of OpenLSTO is to develop efficient and robust tool for multiscale and multiphysics design optimization for engineering structures as well as to provide modular and extensible environment for future development. In that sense, this manual provides an introduction to OpenLSTO principles and internal structures.

The OpenLSTO software suite is composed of two C++ based software modules that perform a wide range of level set based structural topology optimization tasks. An overall description of each module is included below to give perspective on the suite’s capabilities, while more details can be found in the Developer’s Guide. M2DO_FEA can be executed individually to perform finite element analysis, but the real power of the suite lies in the coupling of the modules to perform complex activities, including design optimization.

A key feature of the C++ modules is that each has been designed to separate functionality as much as possible and to leverage the advantages of the class-inheritance structure of the programming language. This makes OpenLSTO an ideal platform for prototyping new numerical methods, discretization schemes, governing equation sets, mesh perturbation algorithms, adaptive mesh refinement schemes, parallelization schemes, etc. You simply need to define a new subclass and get down to business. This philosophy makes OpenLSTO quickly extensible to a wide variety of PDE analyses suited to the needs of the user, and work is ongoing to incorporate additional features for future OpenLSTO releases. The key elements in the OpenLSTO software suite are briefly described below for the current release, but note that modules may be added and removed with future development.
Chapter 2

General Structure

General structure of the code is shown in Figure 2.1. It contains two main modules, including FEA (finite element analysis) and LSM (level set method), to solve a standard structural topology optimization problem through evolving a Design Domain that are represented by Area Fraction Field in the FEA module and Level Set Field in the LSM module. FEA contains and manages lists of degree of freedom managers, elements, boundary conditions, material constitutive properties, and applied boundary conditions for structural analysis. Services for accessing each of these objects are provided. LSM contains and manages list of elements, level set functions, and structural boundary operation functions. Although Optimizer is currently embedded in LSM, it actually serves as an interface between FEA and LSM. Sensitivities of structural response calculated from FEA are input into Optimizer, and solutions for structural boundary movement are output and supplied to LSM to evolve or update the structural boundary. More details will be provided in the following chapters.

![Figure 2.1: General structure of OpenLSTO framework.](image)

---

**Figure 2.1:** General structure of OpenLSTO framework.
An example of key elements for a typical main file scope for implementing OpenL-STO is given below:

```cpp
#include "M2DO_FEA.h"
#include "M2DO_LSM.h"
#include "MatrixM2DO.h"

using namespace std;

namespace FEA = M2DO_FEA;
namespace LSM = M2DO_LSM;

int main() {

    // PROGRAM BODY TEXT

    return 0;
}
```
Chapter 3

Finite Element Analysis Module

In this chapter, the structure of finite element analysis module is deepened. Organization of parameters that belong to this module is also stretched, so the user can get a first touch with the reasoning behind the coding.

The finite element analysis module is decomposed into several classes according to typical procedure of performing a structural analysis, as already presented in Figure 2.1. The main components are:

- class SolidMaterial
- class Mesh
- class HomogeneousDirichletBoundaryConditions
- class StationaryStudy
- class Sensitivity

They are declared in the header file as follows:

```cpp
#ifndef M2DO_FEA_MODULE_H
#define M2DO_FEA_MODULE_H
...

using namespace std;

#include "quadrature.h"
#include "linear_shape_function.h"
#include "node.h"
#include "solid_element.h"
#include "solid_material.h"
#include "mesh.h"
```
An instance of a finite element model is declared as follows:

```cpp
// FEA mesh object for 2D analysis:
FEA::Mesh fea_mesh(2);

// FEA mesh object for 3D analysis:
FEA::Mesh fea_mesh(3);
```

Both 2D and 3D models can be encountered in current version of Open_LSTO code. For this, user needs to define the geometric dimension of the problem when declaring the `Mesh` object.

Material properties, geometry of structure, type of element, meshing, boundary conditions and type of study are then generated through calling corresponding functions, which are given in details in following sections.

### 3.1 Material

`SolidMaterial` represents base class for all constitutive models, once it is used to define the material properties of the structure. The following commands can be used to set the Young’s modulus, Poisson’s ratio and density:

```cpp
double E = 1.0; // Young’s Modulus
double nu = 0.3; // Poisson’s ratio
double rho = 1.0; // Density
fea_mesh.solid_materials.push_back(FEA::SolidMaterial(2, E, nu, rho));
```

Constitutive matrix, $C$, for the given material is computed. Its implementation is carried out as follows:

```cpp
SolidMaterial :: SolidMaterial (int spacedim, double E, double nu, double rho, double h) : spacedim (spacedim), E (E), nu (nu), rho (rho), h (h) {
    if (spacedim == 2) {
        Matrix<double,-1,-1> A(4,4);
    }
```
A.data = {{1, 0, 0, 0},
          {0, 0.5, 0.5, 0},
          {0, 0.5, 0.5, 0},
          {0, 0, 0, 1}};

// Voight matrix:
Matrix<double,-1,-1> V(4,4);
V.data ={{ 1, 0, 0, -0.5 },
        {0, 1.5, 0, 0},
        {0, 0, 1.5, 0},
        {-0.5, 0, 0, 1}};

// Note: This is the plane stress formulation!
Matrix<double,-1,-1> D(4,4);
D.data = {{1, 0, 0, nu},
          {0, (1-nu)/2, (1-nu)/2, 0},
          {0, (1-nu)/2, (1-nu)/2, 0},
          {nu, 0, 0, 1}};
D *= E / (1-pow(nu,2)) ;
C = (D.dot(A));
C *= h;

else if (spacedim == 3) {

Matrix<double,-1,-1> A(9,9);
A.data = {{ 1, 0, 0, 0, 0, 0, 0, 0, 0},
          {0, 0.5, 0.5, 0, 0, 0, 0, 0, 0},
          {0, 0, 0.5, 0, 0, 0, 0.5, 0, 0},
          {0, 0.5, 0, 0.5, 0, 0, 0, 0, 0},
          {0, 0, 0, 0, 1, 0, 0, 0, 0},
          {0, 0, 0, 0, 0, 0.5, 0, 0.5, 0},
          {0, 0, 0, 0, 0, 0.5, 0.5, 0, 0},
          {0, 0, 0, 0, 0, 0, 0, 0, 1}};
Matrix<double, -1, -1> D(9, 9);

D.data = {{{(1-nu), 0, 0, 0, nu, 0, 0, 0, nu},
{0, (1-2*nu), 0, 0, 0, 0, 0, 0, 0},
{0, 0, (1-2*nu), 0, 0, 0, 0, 0, 0},
{nu, 0, 0, 0, (1-nu), 0, 0, 0, nu},
{0, 0, 0, 0, (1-2*nu), 0, 0, 0, 0},
{0, 0, 0, 0, 0, (1-2*nu), 0, 0, 0},
{0, 0, 0, 0, 0, 0, (1-2*nu), 0, 0},
{nu, 0, 0, 0, nu, 0, 0, 0, (1-nu)}};

D *= E / ((1+nu) * (1-2*nu)) ;

C = D.dot(A) ;

}
8 nodes, each having $x$-, $y$- and $z$-direction displacements as degrees of freedom. Elemental stiffness matrix $K_e = B^T C B$ is implemented in \texttt{SolidElement} class and is computed as follows:

```cpp
Matrix<double,-1,-1> SolidElement :: K () {
    Matrix<double,-1,-1> J_mat, J_inv, K_mat(pow(2, spacedim) * spacedim, 
        pow(2, spacedim) * spacedim);  
    Matrix<double,-1,-1> C;  
    C = mesh.solid_materials[material_id].C;  
    Vector<double,-1> shape_grad_j, shape_grad_j_full;  

    K_mat.fill(0.0);  
    double w;  
    vector<double> eta (spacedim, 0), eta_count (spacedim, 0);  

    /*
    grad(u(x)) = [B] * \{u\}
    */
    Matrix<double,-1,-1> B_mat(spacedim * spacedim, pow(2, spacedim) * 
        spacedim);  
    int n_gauss = pow (order, spacedim);  
    for (int k = 0 ; k < n_gauss ; ++k) {
        int shape_j = 0, dim_j = 0;  

        /*
        The first gauss point is located at eta = [quadrature.eta[0],
            quadrature.eta[0], ...].
        Since eta_count was initialized with zeros in all dimensions, we
don't need to do
        anything fancy yet; just set eta[1] =
        quadrature.eta[eta_count[1]], etc. Same goes
        for the weighting w.
        */

        w = 1.0;
    }
}
```
for (int l = 0; l < spacedim; ++l) {
    eta[l] = quadrature.eta[eta_count[l]];
    w *= quadrature.w[eta_count[l]];
}

J_mat = J(eta);
J_inv = J_mat.inverse();

/*
  Build the B matrix at this gauss point:
*/

for (int j = 0; j < spacedim * pow(2, spacedim); ++j) {
    shape_grad_j =
        J_inv.dot(linear_shape_function.GetShapeFunctionGradientsVector(shape_j, eta));
    shape_grad_j_full =
        linear_shape_function.GetShapeFunctionGradientsFullVector(shape_grad_j, dim_j);

    // hac210 note: col.
    // B_mat.col(j) = shape_grad_j_full;
    for (int kk = 0; kk < B_mat.rows(); kk ++) {
        B_mat(kk, j) = shape_grad_j_full(kk);
    }

    if (dim_j < spacedim - 1) {
        dim_j = dim_j + 1;
    } else {
        dim_j = 0;
        shape_j = shape_j + 1;
    }

} // for j (columns in B).

eta_count = quadrature.UpdateEtaCounter(eta_count);
The corresponding strain-displacement matrix, $B$, is accomplished as:

```cpp
Matrix<double,-1,-1> SolidElement :: B (vector<double> & eta) {
    Vector<double,-1> shape_grad_j, shape_grad_j_full;
    
    Matrix<double,-1,-1> B_mat(spacedim * spacedim, pow(2, spacedim) * spacedim);
    B_mat.fill(0.0);
    
    int shape_j = 0, dim_j = 0;
    
    Matrix<double,-1,-1> J_mat = J (eta);
    Matrix<double,-1,-1> J_inv = J_mat.inverse();
```
// Build the B matrix at the given eta point:
for (int j = 0 ; j < spacedim * pow(2, spacedim) ; ++j) {

    shape_grad_j =
        J_inv.dot(linear_shape_function.GetShapeFunctionGradientsVector(shape_j, eta)) ;
    shape_grad_j_full =
        linear_shape_function.GetShapeFunctionGradientsFullVector(shape_grad_j, dim_j) ;

    // hac210 note: col.
    // B_mat.col(j) = shape_grad_j_full ;
    for (int kk = 0; kk < B_mat.rows(); kk ++){
        B_mat(kk,j) = shape_grad_j_full(kk) ;
    }

    if (dim_j < spacedim-1) {
        dim_j = dim_j + 1 ;
    }

    else {
        dim_j = 0 ;
        shape_j = shape_j + 1 ;
    }
}
// for j (columns in B).
return B_mat ;

3.2.2 Meshing

To discretize a 2D design domain, a mesh with quadrilateral elements is generated via method MeshSolidHyperRectangle. The degrees of freedom are assigned to each node by AssignDof. Such standard mesh can be initialized using following commands:

// Element Gauss integration order:
int element_order = 2 ;
// Create structured mesh and assign degrees of freedom:
fea_mesh.MeshSolidHyperRectangle ([nelx, nely], fea_box, element_order, false);
fea_mesh.is_structured = true;
fea_mesh.AssignDof();

where `element_order` assigns the number of Gaussian integration points, e.g. an `element_order` of 2 uses a total of 4 Gauss points.

`MeshSolidHyperRectangle` discretizes a 2D geometry into rectangular elements. Both elements and nodes are enumerated from left to right and from bottom to top. In each element, the four nodes are enumerated from the left bottom corner one to the left top corner one in the counter-clockwise direction. Nodes are collected into the `Node` class, where nodal information includes coordinates, nodal number and degrees of freedom are stored. The implementation of `MeshSolidHyperRectangle` is given in the following snippet of the code:

```cpp
void Mesh :: MeshSolidHyperRectangle (vector<int> nel,
    Matrix<double,-1,-1> mesh_box, int element_order, bool time_it) {

    auto t_start = chrono::high_resolution_clock::now() ;

    if (time_it) {
        cout << "\nMeshing solid hyper-rectangle ... " << flush ;
    }

    /*
    * First mesh the natural hyper-rectangle. We can utilise a sequence
    * similar to that in quadrature.UpdateEtaCounter() for this.
    */

    int num_nodes = 1, num_elements = 1 ;

    for (int i = 0 ; i < nel.size() ; ++i) {
        num_nodes *= nel[i]+1 ;
        num_elements *= nel[i] ;
    }

    nodes.reserve (num_nodes);
```
solid_elements.reserve (num_elements) ;

/*
   Each of these linear elements has pow (2, spacedim) nodes;
   each node has dim degrees of freedom. Using this we can
   calculate num_entries (needed for study step).
*/

// num_entries += num_elements * pow(pow(2, spacedim) * dim), 2) ;
// n_dof = num_nodes * dim ;

vector<int> eta_count (spacedim, 0) ;

for (int i = 0 ; i < num_nodes ; ++i) {

   Node node (spacedim) ;
   node.id = i ;

   for (int l = 0 ; l < spacedim ; ++l) {

      node.coordinates[l] = -1 + eta_count[l] * (2.0 / nel[l]) ;

   }

   nodes.push_back(node) ;

   // Update eta_count:
   eta_count[0] += 1 ;

   if (eta_count[0] > nel[0]) {

      eta_count[0] = 0 ;

      for (int l = 1 ; l < spacedim ; ++l) {

         eta_count[l] += 1 ;

         if (eta_count[l] <= nel[l]) {
            break ;
         }
      }
else {
    eta_count[l] = 0;
}

for (int l = 1; l < spacedim; ++l) {
    nel_mult[l] *= nel_mult[l-1] * (nel[l-1]+1);
}

SolidElement element (spacedim, element_order, *this);  
for (int i = 0; i < num_elements; ++i) {
    start_id = 0;
    for (int l = 0; l < spacedim; ++l) {
        start_id += nel_count[l] * nel_mult[l];
    }
}
// This just fills the node_ids vector with start_id:
fill(node_ids.begin(), node_ids.end(), start_id);

for (int j = 0; j < pow(2, spacedim); ++j) {
    eta = linear_shape_function.GetEta(j);

    // Change the -1 values to zeros. Also, eta
    // comes as doubles so change to int, then
    // multiply by nel_mult:
    for (int k = 0; k < spacedim; ++k) {
        eta_int = (eta[k] < 0) ? 0 : 1;
        node_ids[j] += eta_int * nel_mult[k];
    }
    element.node_ids[j] = node_ids[j];
}

// Add the element to the mesh:
solid_elements.push_back(element);

// Update nel_count:
nel_count[0] += 1;

if (nel_count[0] > nel[0]-1) {
    nel_count[0] = 0;
    for (int l = 1; l < spacedim; ++l) {
        nel_count[l] += 1;
        if (nel_count[l] < nel[l]) {
        
        
        }
break ;
}

else {
    nel_count[l] = 0 ;
}
}
}

/*
Now we deform the natural mesh to conform to the physical
geometry using linear shape functions:

x = sum(N_i * x_i) etc. where in this case x_i are the
coordinates of the corners of the hyper_rectangle to mesh.
*/

Vector<double,-1> shape_value_vec ;

for (int i = 0 ; i < nodes.size() ; ++i) {
    // Shape function values at natural coordinate:
    shape_value_vec =
        linear_shape_function.GetShapeFunctionValuesVector(nodes[i].coordinates)
        ;

    for (int j = 0 ; j < spacedim ; ++j) {
        // hac210 note: cols
        nodes[i].coordinates[j] = 0.0;

        for (int pp = 0; pp < mesh_box.rows(); pp ++){
            nodes[i].coordinates[j] += mesh_box(pp,j)*shape_value_vec(pp) ;
        }
    }
}
auto t_end = chrono::high_resolution_clock::now();

if (time_it) {
    cout << "Done. Time elapsed = " << 
         chrono::duration<double>(t_end-t_start).count() << "\n" << flush ;
}

Each node is assigned to specific degrees of freedom according to the underlying type of element. In general, the order of degrees of freedom is based on the order of nodes. For instance, the $i$-th node for a plane stress rectangular element has $2i$ and $2i+1$ degrees of freedom. The implementation is given below:

```cpp
void Mesh::AssignDof() {
    n_dof = 0;

    /*
    Solid elements: these have spacedim displacement dofs per node, and there are pow(2, spacedim) nodes per element.
    */
    for (auto &&element : solid_elements) {
        element.dof = vector<int> (spacedim * pow(2, spacedim), -1);
        for (int i = 0; i < element.node_ids.size(); ++i) {
            auto &&node = nodes[element.node_ids[i]];
            /*
            Check if this node has already assigned a dof number; If not, assign one:
            */
            for (int j = 0; j < spacedim; ++j) {
                if (node.dof[j] >= 0) {
```
element.dof [i*spacedim + j] = node.dof [j] ;

} else {
    element.dof [i*spacedim + j] = n_dof ;
    node.dof [j] = n_dof ;
    n_dof += 1 ;

}

} // for j (spacedim).

} // for i (element.node_ids.size()).

} // for element : solid_elements.


3.3 Analysis Type

Current version of Open_LSTO code provides structural static analysis. Future releases will bring more types of analysis, such as conduction heat transfer and thermostructural finite element analysis.

To declare a static analysis, the class FEA::StationaryStudy is used and implemented as follows:

FEA::StationaryStudy fea_study (fea_mesh) ; // Initialize study

Functions such as assembling global stiffness matrices and force vector, applying boundary conditions and solving the equilibrium equation ([K]{u} = {f}) are called to set up the analysis.

3.3.1 Boundary Conditions

Boundary conditions application is a very important step on the finite element analysis procedure. In that sense, following lines bring the implementation of Dirichlet and Neumann boundary conditions within the framework of Open_LSTO code. For
Figure 3.1: Configuration of the cantilever beam subject to a point load.

For more details on the manipulation of boundary conditions, the reader is referred to the *Tutorial for OpenLSTO v0.2: Open Source Level Set Topology Optimization* manual.

**Dirichlet Boundary Conditions**

To apply Dirichlet boundary conditions, degrees of freedom to be constrained are selected. The free DOF’s are grouped into a set as reduced DOF’s and they are managed through the `M2DO_FEA::HomogeneousDirichletBoundaryConditions` class. Constraints on the degrees of freedom are imposed through `AddBoundaryConditions` in the `StationaryStudy` class.

Consider a cantilever beam depicted in Figure 3.1 as example, where left edge is fixed. Displacement boundary conditions need to be applied on the left-hand side of the domain. This can be done in the following way:

```cpp
  // Select the appropriate nodes and DOFs corresponding to the left hand side of domain
  vector<double> coord = {0.0, 0.0}, tol = {1e-12, 1e10};
  vector<int> fixed_nodes = fea_mesh.GetNodesByCoordinates (coord, tol);
  vector<int> fixed_dof = fea_mesh.dof (fixed_nodes);

  // Add boundary conditions to study:
  fea_study.AddBoundaryConditions
      (FEA::HomogeneousDirichletBoundaryConditions (fixed_dof,
          fea_mesh.n_dof));
```

**Neumann Boundary Conditions**

Forces applied to a given set of nodes are carried by selecting corresponding DOF’s and assigning their magnitudes. In current code, only point loads are permitted to be applied.
For instance, in the cantilever beam depicted in Figure 3.1, a vertical point load is applied at the mid point of the right-hand side of the domain using the following commands:

```cpp
// Select dof using a box centered at coord of size tol:
coord = {1.0*nelx, 0.5*nely}, tol = {1e-12, 1e-12};
vector<int> load_node = fea_mesh.GetNodesByCoordinates(coord, tol);
vector<int> load_dof = fea_mesh.dof(load_node);

vector<double> load_val(load_node.size() * 2);
for (int i = 0; i < load_node.size(); ++i) {
    load_val[2*i] = 0.00; // load component in x direction.
    load_val[2*i+1] = -0.5; // load component in y direction.
}
// Add point load to study and assemble load vector {f}:
FEA::PointValues point_load(load_dof, load_val);
fea_study.AssembleF(point_load, false);
```

### 3.3.2 Assembling Global Stiffness Matrix

The finite element analysis is carried out using the area fraction method, where the stiffness matrix of an element is computed using the formula below:

\[
K_e = x_i K_0^e
\]  

(3.1)

Here, \(K_e\) is the elemental stiffness matrix, \(x_i\) is the element area fraction of the solid portion determined by the level set and \(K_0^e\) is the elemental stiffness calculated assuming \(x_i = 1\).

The global stiffness matrix is assembled in a sparse way, this means that only the non-zero values of the global stiffness matrix and their respective row and column numbers are saved. For this purpose, a data structure called `Triplet_Sparse` is used. `Triplet_Sparse` has the following members: `row`, `col` and `val`, which represent the row, column and the non-zero value. For example, if \(K(i,j) = v\), then the sparse representation for this value would be:

```cpp
Triplet_Sparse.row = i;
Triplet_Sparse.col = j;
Triplet_Sparse.val = v;
std::vector<Triplet_Sparse> K_global; // stiffness matrix
```

Thus, the stiffness matrix is just a vector containing `Triplet_Sparse` data.
The conjugate gradient method (StationaryStudy::Solve_With.CG) is used to solve the system of linear equations defined as \([K]{u} = \{f\}\), where \([K]\) is the reduced global stiffness matrix, \({u}\) is the reduced global displacement vector and \({f}\) is the reduced global load vector. The algorithm is described as follows:

- set initial displacement: \(u_0 = 0\)
- set the initial residual: \(r_0 = \{f\} - [K] \{u_0\}\)
- set the conjugate gradient: \(p_0 = r_0\)
- set the iteration counter \(k = 0\)
- while \(||r_k|| > tol||\), do
  \[\alpha_k = \frac{r_k^T r_k}{p_k^T K p_k}\]
  \[u_{k+1} = u_k + \alpha_k p_k\]
  \[r_{k+1} = r_k - \alpha_k K p_k\]
  \[\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}\]
  \[p_{k+1} = r_{k+1} + \beta_k p_k\]
  \[k = k + 1\]

The most time-consuming part of the conjugate gradient algorithm is the matrix-vector multiplication. The method StationaryStudy::mat_vec_mult is used to efficiently compute the sparse-matrix-vector multiplication and is implemented as follows:

```cpp
void StationaryStudy::mat_vec_mult( vector<Triplet_Sparse> &K, 
     vector<double> &v_in, vector<double> &v_out ){
  #pragma omp parallel for
  for(int i = 0; i < K.size(); i++) {
    v_out[K[i].row] += K[i].val*v_in[K[i].col];
  }
}
```

### 3.3.3 Sensitivity Analysis

The class FEA::SensitivityAnalysis helps to compute sensitivities based on the finite element solution. The function ComputeComplianceSensitivities calculates the compliance shape sensitivity at each gauss point of the finite element mesh. ComputeBoundarySensitivities(boundary_point) calculates shape sensitivities at a given boundary point by using the weighted least-squares interpolation. The following commands implement the sensitivity calculation:
FEA::SensitivityAnalysis sens (fea_study); // create a sensitivity object

// Compute compliance sensitivities (stress*strain) at the Gauss points:
sens.ComputeComplianceSensitivities (false);

// define a boundary point
vector<double> boundary_point (2, 0.0);

// Interpolate Gauss point sensitivities by using least-squares method
sens.ComputeBoundarySensitivities (boundary_point);
Chapter 4

Level Set Method Module

Level set method module consists of the following classes:

- Mesh
- LevelSet
- Boundary
- FastMarchingMethod
- Heap
- Hole
- Optimise
- InputOutput

They are declared in the M2DO_LSM header file as described below:

```cpp
#ifndef M2DO_LSM_MODULE_H
#define M2DO_LSM_MODULE_H
...

// Should put these in the namespace!
#include "debug.h"
#include "min_unit.h"

namespace M2DO_LSM {
  #include "common.h"
  #include "mesh.h"
}
```
4.1 Create a Level Set Field

The following snippet of code gives a typical procedure to create a level set field for representing the topology of a structure:

```cpp
#include "hole.h"
#include "heap.h"
#include "fast_marching_method.h"
#include "level_set.h"
#include "boundary.h"
#include "input_output.h"
#include "optimise.h"
#include "sensitivity.h"

}
#endif

4.1 Create a Level Set Field

The following snippet of code gives a typical procedure to create a level set field for representing the topology of a structure:

```cpp
// Seed initial holes:
vector<LSM::Hole> holes ;
holes.push_back (LSM::Hole (16, 14, 5)) ;

// Initialise the level set mesh (same resolution as the FE mesh):
LSM::Mesh lsm_mesh (nelx, nely, false) ;

// Initialise the level set object (from the hole vector):
LSM::LevelSet level_set (lsm_mesh, holes, move_limit, band_width, is_fixed_domain) ;

// Reinitialise the level set to a signed distance function:
level_set.reinitialise () ;
```

It uses the classes `Hole`, `Mesh` and `LevelSet`. A level set mesh of \( nelx \times nely \) is generated to create a mesh grid for the structure and some initial topologies features, such as circular holes, are inserted into the mesh grid through `Hole`. An instance of the level set field is declared by calling `LevelSet` and the signed distance field is calculated through a fast marching method. More details of each these classes are given in the following sections.
### 4.1.1 Generate Level Set Mesh

`Mesh::Mesh` discretizes a geometry into rectangular elements. Elements and nodes are enumerated from left to right and from bottom to top through subroutines `Mesh::initialiseNodes` and `Mesh::initialiseElements`. In each element the four nodes are enumerated from the left bottom corner one to the left top corner one in the counter-clockwise direction. Features of a node, which includes the coordinates, neighbour nodes, connected elements, among others, are stored in a `Struct Node`. Element related attributes, such as coordinate of elemental centroid and indices of nodes of the element are collected in a `Struct Element`. The implementation of `Mesh::Mesh` is done as follows:

```cpp
Mesh::Mesh(unsigned int width_,
          unsigned int height_,
          bool isPeriodic_)
{
    // Resize element and node data structures.
    elements.resize(nElements);
    nodes.resize(nNodes);

    // Resize 2D to 1D mapping vector.
    xyToIndex.resize(width+1);
    for (unsigned int i=0;i<width+1;i++)
        xyToIndex[i].resize(height+1);

    // Calculate node nearest neighbours.
    initialiseNodes();

    // Initialise elements (and node to element connectivity).
    initialiseElements();
}
```
4.1.2 Calculate Signed Distance Field

For the created level set mesh, signed distances of each mesh grid to boundaries of a structure can be calculated through LevelSet::closestDomainBoundary, which computes the distance to the closest boundary as showed below:

```cpp
void LevelSet::closestDomainBoundary()
{
    // Initial LSF is distance from closest domain boundary.
    for (unsigned int i=0;i<mesh.nNodes;i++)
    {
        // Closest edge in x.
        unsigned int minX = std::min(mesh.nodes[i].coord.x, mesh.width - mesh.nodes[i].coord.x);

        // Closest edge in y.
        unsigned int minY = std::min(mesh.nodes[i].coord.y, mesh.height - mesh.nodes[i].coord.y);

        // Signed distance is the minimum of minX and minY;
        signedDistance[i] = double(std::min(minX, minY));
    }
}
```

When additional features, such as circular holes, are introduced, the signed distance field needs to be updated accordingly. Signed distances from each node to the inserted holes are computed and boolean operations are conducted to determine the shortest distance from the node to the closest boundary. Hence, a level set field is initialised for the design domain. It is implemented as the following lines:

```cpp
void LevelSet::initialise(const std::vector<Hole>& holes)
{
    // First initialise the signed distance based on domain boundary.
    closestDomainBoundary();

    /* Now test signed distance against the surface of each hole.
     * Update signed distance function when distance to hole surface
     * is less than the current value. Since this is only done once, we
     * use the simplest implementation possible.
     */

    // Loop over all nodes.
    for (unsigned int i=0;i<mesh.nNodes;i++)
    {
```
4.2 Boundary Discretization

Only the signed distance for mesh grid is computed when initializing it. However, the intersection points between the boundary and the mesh grid are required in the further calculations, such as computing elemental area fraction and movements of boundary. The following code illustrates the key procedure for boundary discretization:

```cpp
// Initialise the boundary object :
LSM::Boundary boundary (level_set) ;
// Perform boundary discretisation:
boundary.discretise (false, lambdas.size()) ;
// Compute element area fractions:
boundary.computeAreaFractions () ;
```

The boundary intersection points are computed by looking for nodes lying exactly on the zero contour of the level set and then constructing a set of additional boundary points by simple linear interpolation when the level set changes sign between the nodes on an element edge. This is implemented in the subroutine
Boundary::discretise. The points vector holds coordinates for boundary points (both those lying exactly on nodes of the level set mesh and the interpolated points), which are collected through BoundaryPoints. Boundary segment data is stored in the segments vector. With the obtained boundary points, element cut by the boundary can be computed by using Marching Square or Cube algorithms for 2D or 3D problems, respectively. For instance, Marching Square algorithm for 2D problem is implemented in cutArea. The material area for an element cut by the boundary in terms of area fraction is then obtained from computeAreaFractions. The vector of area fractions is past to finite element analysis module to conduct the fixed grid finite element analysis. Following lines show the computational implementation of these routines:

```cpp
//! Calculate the material area for an element cut by the boundary.
/*! \param element 
A reference to the element.

\return The area fraction.
*/
double Boundary::cutArea(const Element& element)
{
  // Number of polygon vertices.
  unsigned int nVertices = 0;

  // Polygon vertices (maximum of six).
  std::vector<Coord> vertices(6);

  // Whether we're searching for nodes that are inside or outside the boundary.
  NodeStatus::NodeStatus status;

  if (element.status & ElementStatus::CENTRE_OUTSIDE) status = NodeStatus::OUTSIDE;
  else status = NodeStatus::INSIDE;

  // Check all nodes of the element.
  for (unsigned int i=0;i<4;i++)
  {
    // Node index;
    unsigned int node = element.nodes[i];
```
// Node matches status.
if (levelSet.mesh.nodes[node].status & status)
{
    // Add coordinates to vertex array.
    vertices[nVertices].x = levelSet.mesh.nodes[node].coord.x;
    vertices[nVertices].y = levelSet.mesh.nodes[node].coord.y;

    // Increment number of vertices.
    nVertices++;
}

// Node is on the boundary.
else if (levelSet.mesh.nodes[node].status & NodeStatus::BOUNDARY)
{
    // Next node.
    unsigned int n1 = (i == 3) ? 0 : (i + 1);
    n1 = element.nodes[n1];

    // Previous node.
    unsigned int n2 = (i == 0) ? 3 : (i - 1);
    n2 = element.nodes[n2];

    // Check that node isn’t part of a boundary segment, i.e. both
    // neighbours are inside the structure.
    if ((levelSet.mesh.nodes[n1].status & NodeStatus::INSIDE) &&
        (levelSet.mesh.nodes[n2].status & NodeStatus::INSIDE))
    {
        // Add coordinates to vertex array.
        vertices[nVertices].x = levelSet.mesh.nodes[node].coord.x;
        vertices[nVertices].y = levelSet.mesh.nodes[node].coord.y;

        // Increment number of vertices.
        nVertices++;
    }
}

// Add boundary segment start and end points.
for (unsigned int i=0;i<element.nBoundarySegments;i++)
{

// Segment index.
unsigned int segment = element.boundarySegments[i];

// Add start point coordinates to vertices array.
vertices[nVertices].x = points[segments[segment].start].coord.x;
vertices[nVertices].y = points[segments[segment].start].coord.y;

// Increment number of vertices.
nVertices++;

// Add end point coordinates to vertices array.
vertices[nVertices].x = points[segments[segment].end].coord.x;
vertices[nVertices].y = points[segments[segment].end].coord.y;

// Increment number of vertices.
nVertices++;
}

// Return area of the polygon.
if (element.status & ElementStatus::CENTRE_OUTSIDE)
    return (1.0 - polygonArea(vertices, nVertices, element.coord));
else
    return polygonArea(vertices, nVertices, element.coord);
}

//! Calculate the material area fraction in each element.
/*! 
 \return The total element area fraction.
 */
double Boundary::computeAreaFractions()
{
    // Zero the total area fraction.
    area = 0;

    for (unsigned int i=0;i<levelSet.mesh.nElements;i++)
    {
        // Element is inside structure.
        if (levelSet.mesh.elements[i].status & ElementStatus::INSIDE)
            levelSet.mesh.elements[i].area = 1.0;

        // Element is outside structure.
    }
else if (levelSet.mesh.elements[i].status & ElementStatus::OUTSIDE)
    levelSet.mesh.elements[i].area = 0.0;

    // Element is cut by the boundary.
    else levelSet.mesh.elements[i].area =
        cutArea(levelSet.mesh.elements[i]);

    // Add the area to the running total.
    area += levelSet.mesh.elements[i].area;

} return area;

4.3 Evolve Level Set Field

Hamilton-Jacob equation is solved to update the level set field, and its implementation is described here. Users are referred to read <Theoretical Background> for details.

// Initialize optimise class
LSM::Optimise optimise (boundary.points, time_step, move_limit) ;
// set up required parameters
optimise.length_x = lsm_mesh.width ;
optimise.length_y = lsm_mesh.height ;
optimise.boundary_area = boundary.area ; // area of structure
optimise.mesh_area = mesh_area ; // area of the entire mesh
optimise.max_area = max_area ; // maximum area, i.e. area constraint
// Perform the optimisation
optimise.Solve_With_NewtonRaphson () ;
// Extend boundary point velocities to all narrow band nodes
level_set.computeVelocities (boundary.points, time_step, 0, rng) ;
// Compute gradient of the signed distance function within the narrow band
level_set.computeGradients () ;
// Update the level set function
bool is_reinitialised = level_set.update (time_step) ;
// Reinitialise the signed distance function, if necessary
level_set.reinitialise () ;
4.3.1 Optimization

The optimization is carried out in the `LSM::Optimise` class, which is a lightweight class initialized in the iteration loop itself. This instantiates a lightweight object and the cost associated to the reinitialisation of it at every iteration is considered low. The optimization problem can be linearised at every iteration and is described as follows:

\[
\begin{align*}
\text{Minimize} & \quad C = C_0 + \sum_i^N C_i^f z_i \\
\text{subject to} & \quad A = A_0 + \sum_i^N C_i^g z_i \leq A_{\text{max}} \\
& \quad z_{\text{min}} \leq z_i \leq z_{\text{max}} \\
& \quad z_i = \lambda_f s_i^f + \lambda_g s_i^g; \quad i = 1...N \quad (4.1)
\end{align*}
\]

where

- \( z_i \) is the displacement of the \( i^{th} \) boundary point;
- \( C \) is the compliance, linearized around \( C_0 \) (current compliance);
- \( A \) is the area, linearized around \( A_0 \) (current area);
- \( N \) is the number of boundary points;
- \( C_i^f = s_i^f l_i \) is the sensitivity \( (s_i^f) \) of the compliance w.r.t. the \( i^{th} \) boundary point, multiplied by segment length \( (l_i) \);
- \( C_i^g = s_i^g l_i \) is the sensitivity \( (s_i^g) \) of the area w.r.t. the \( i^{th} \) boundary point, multiplied by the segment length \( (l_i) \);
- \( z_{\text{max}} \) and \( z_{\text{min}} \) are the maximum and minimum allowed displacement for the boundary point, respectively;
- \( \lambda_f \) is the Lagrangian multiplier associated with the objective function;
- \( \lambda_g \) is the Lagrangian multiplier associated with the constraint (area).

By setting \( \lambda_g \) equal to the `move_limit`, the optimization problem Eq. 4.1 becomes a simple one dimensional optimization problem in the variable \( (\lambda_f) \), that can be solved by using the Newton-Raphson method.
4.3.2 Velocity

Having obtained $\lambda_f$, the optimum velocities at the boundary points can be calculated. The structural boundary can hence be updated to find the new structure. In order to update the level set function, velocity values are required at all grid points. The velocity values are extended from boundary points to grid points by solving the following equation:

$$\nabla \phi \cdot \nabla V = 0$$  \hspace{1cm} (4.2)

where $V$ is the velocity field at the grid points. Eq. 4.2 is solved using the Fast Marching Method. The level set is then updated by solving the following advection equation:

$$\frac{\partial \phi}{\partial t} + V|\nabla \phi| = 0.$$  \hspace{1cm} (4.3)

The snippet of code within the Level Set Method Module that performs aforementioned routines is given below:

```cpp
//! Extend boundary point velocities to the level set nodes.
/*! \param boundaryPoints
A reference to a vector of boundary points.
*/
void LevelSet::computeVelocities(const std::vector<BoundaryPoint>& boundaryPoints)
{
    // Initialise velocity (map boundary points to boundary nodes).
    initialiseVelocities(boundaryPoints);

    // Initialise fast marching method object.
    FastMarchingMethod fmm(mesh, false);

    // Reinitialise the signed distance function.
    fmm.march(signedDistance, velocity);
}

//! Initialise velocities for boundary nodes.
/*! \param boundaryPoints
A reference to a vector of boundary points.
*/
void LevelSet::initialiseVelocities(const std::vector<BoundaryPoint>& boundaryPoints)
{
    // Map boundary point velocities to nodes of the level set domain
    // using inverse squared distance interpolation.
```
// Whether the velocity at a node has been set.
bool isSet[mesh.nNodes];

// Weighting factor for each node.
double weight[mesh.nNodes];

// Initialise arrays.
for (unsigned int i=0;i<mesh.nNodes;i++)
{
    isSet[i] = false;
    weight[i] = 0;
    velocity[i] = 0;
}

// Loop over all boundary points.
for (unsigned int i=0;i<boundaryPoints.size();i++)
{
    // Find the closest node.
    unsigned int node = mesh.getClosestNode(boundaryPoints[i].coord);

    // Distance from the boundary point to the node.
    double dx = mesh.nodes[node].coord.x - boundaryPoints[i].coord.x;
    double dy = mesh.nodes[node].coord.y - boundaryPoints[i].coord.y;

    // Squared distance.
    double rSqd = dx*dx + dy*dy;

    // If boundary point lies exactly on the node, then set velocity
    // to that of the boundary point.
    if (rSqd < 1e-6)
    {
        velocity[node] = boundaryPoints[i].velocity;
        weight[node] = 1.0;
        isSet[node] = true;
    }
    else
    {
        // Update velocity estimate if not already set.
        if (!isSet[node])
        {
            
        }
    }
}
velocity[node] += boundaryPoints[i].velocity / rSqd;
weight[node] += 1.0 / rSqd;
}
}

// Loop over all neighbours of the node.
for (unsigned int j=0; j<4; j++)
{
    // Index of the neighbouring node.
    unsigned int neighbour = mesh.nodes[node].neighbours[j];

    // Make sure neighbour is in bounds.
    if (neighbour < mesh.nNodes)
    {
        // Distance from the boundary point to the node.
        double dx = mesh.nodes[neighbour].coord.x -
                    boundaryPoints[i].coord.x;
        double dy = mesh.nodes[neighbour].coord.y -
                    boundaryPoints[i].coord.y;

        // Squared distance.
        double rSqd = dx*dx + dy*dy;

        // If boundary point lies exactly on the node, then set velocity
        // to that of the boundary point.
        if (rSqd < 1e-6)
        {
            velocity[neighbour] = boundaryPoints[i].velocity;
            weight[neighbour] = 1.0;
            isSet[neighbour] = true;
        }
        else if (rSqd <= 1.0)
        {
            // Update velocity estimate if not already set.
            if (!isSet[neighbour])
            {
                velocity[neighbour] += boundaryPoints[i].velocity / rSqd;
                weight[neighbour] += 1.0 / rSqd;
            }
        }
4.3.3 Gradient

Eq. 4.3 is solved numerically by using the forward Euler discretization in time. The spatial gradients at the grid points are computed by using the fifth-order Hamilton-Jacobi Weighted Non-Oscillatory scheme as follows:

```c++
//! Compute the modulus of the gradient of the signed distance function at a node.
/*! 
\param node
   The node index.
*/
double LevelSet::computeGradient(const unsigned int node)
{
   // Nodal coordinates.
   unsigned int x = mesh.nodes[node].coord.x;
   unsigned int y = mesh.nodes[node].coord.y;

   // Nodal signed distance.
   double lsf = signedDistance[node];

   // Whether gradient has been computed.
   bool isGradient = false;

   // Zero the gradient.
   double grad = 0;
   // Compute interpolated velocity.
   for (unsigned int i=0;i<nNarrowBand;i++)
   {
      unsigned int node = narrowBand[i];
      if (velocity[node]) velocity[node] /= weight[node];
   }
```
// Node is on the left edge.
if (x == 0)
{
    // Node is at bottom left corner.
    if (y == 0)
    {
        // If signed distance at nodes to right and above is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
             1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y+1]] - lsf) <
             1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x+1][y+1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }
    // Node is at top left corner.
    else if (y == mesh.height)
    {
        // If signed distance at nodes to right and below is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
             1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y-1]] - lsf) <
             1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x+1][y-1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }
}

// Node is on the left edge.
if (x == 0)
{

    // Node is at bottom left corner.
    if (y == 0)
    {
        // If signed distance at nodes to right and above is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
             1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y+1]] - lsf) <
             1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x+1][y+1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }
    // Node is at top left corner.
    else if (y == mesh.height)
    {
        // If signed distance at nodes to right and below is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
             1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y-1]] - lsf) <
             1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x+1][y-1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }
}

// Node is on the left edge.
if (x == 0)
{
    // Node is at bottom left corner.
    if (y == 0)
    {
        // If signed distance at nodes to right and above is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
             1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y+1]] - lsf) <
             1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x+1][y+1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }
    // Node is at top left corner.
    else if (y == mesh.height)
    {
        // If signed distance at nodes to right and below is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
             1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y-1]] - lsf) <
             1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x+1][y-1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }
}
else if (x == mesh.width) {

    // Node is at bottom right corner.
    if (y == 0) {

        // If signed distance at nodes to left and above is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x-1][y]] - lsf) <
            1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y+1]] - lsf) <
            1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x-1][y+1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }

    // Node is at top right corner.
    else if (y == mesh.height) {

        // If signed distance at nodes to left and below is the same,
        // then use
        // the diagonal node for computing the gradient.
        if ((std::abs(signedDistance[mesh.xyToIndex[x-1][y]] - lsf) <
            1e-6) &&
            (std::abs(signedDistance[mesh.xyToIndex[x][y-1]] - lsf) <
            1e-6))
        {
            // Calculate signed distance to diagonal node.
            grad = std::abs(lsf -
                            signedDistance[mesh.xyToIndex[x-1][y-1]]);
            grad *= sqrt(2.0);
            isGradient = true;
        }
    }
if (!isGradient)
{
    // Stencil values for the WENO approximation.
    double v1, v2, v3, v4, v5;

    // Upwind direction.
    int sign = velocity[node] < 0 ? -1 : 1;

    // Derivatives to right.

    // Node on left-hand edge.
    if (x == 0)
    {
        v1 = signedDistance[mesh.xyToIndex[3][y]] -
            signedDistance[mesh.xyToIndex[2][y]];  
        v2 = signedDistance[mesh.xyToIndex[2][y]] -
            signedDistance[mesh.xyToIndex[1][y]];  
        v3 = signedDistance[mesh.xyToIndex[1][y]] -
            signedDistance[mesh.xyToIndex[0][y]];  

        // Approximate derivatives outside of domain.
        v4 = v3;  
        v5 = v3;
    }

    else if (x == 1)
    {
        v1 = signedDistance[mesh.xyToIndex[4][y]] -
            signedDistance[mesh.xyToIndex[3][y]];  
        v2 = signedDistance[mesh.xyToIndex[3][y]] -
            signedDistance[mesh.xyToIndex[2][y]];  
        v3 = signedDistance[mesh.xyToIndex[2][y]] -
            signedDistance[mesh.xyToIndex[1][y]];  
        v4 = signedDistance[mesh.xyToIndex[1][y]] -
            signedDistance[mesh.xyToIndex[0][y]];  
    }
// Approximate derivatives outside of domain.
    v5 = v4;
}

// Node on right-hand edge.
else if (x == mesh.width)
{
    v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
         signedDistance[mesh.xyToIndex[x-2][y]];
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
         signedDistance[mesh.xyToIndex[x-1][y]];

    // Approximate derivatives outside of domain.
    v3 = v4;
    v2 = v4;
    v1 = v4;
}

// One node to left of right-hand edge.
else if (x == (mesh.width - 1))
{
    v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
         signedDistance[mesh.xyToIndex[x-2][y]];
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
         signedDistance[mesh.xyToIndex[x-1][y]];
    v3 = signedDistance[mesh.xyToIndex[x+1][y]] -
         signedDistance[mesh.xyToIndex[x][y]];

    // Approximate derivatives outside of domain.
    v2 = v3;
    v1 = v3;
}

// Two nodes to left of right-hand edge.
else if (x == (mesh.width - 2))
{
    v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
         signedDistance[mesh.xyToIndex[x-2][y]];
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
         signedDistance[mesh.xyToIndex[x-1][y]];
    v3 = signedDistance[mesh.xyToIndex[x+1][y]] -
         signedDistance[mesh.xyToIndex[x][y]];

    // Approximate derivatives outside of domain.
    v2 = v3;
    v1 = v3;
}
signedDistance[mesh.xyToIndex[x][y]];

v2 = signedDistance[mesh.xyToIndex[x+2][y]] -
    signedDistance[mesh.xyToIndex[x+1][y]];

// Approximate derivatives outside of domain.

// Node lies in bulk.
else
{
    v1 = v2;
}

// Node lies in bulk.

else
{
    v1 = signedDistance[mesh.xyToIndex[x+3][y]] -
        signedDistance[mesh.xyToIndex[x+2][y]];
    v2 = signedDistance[mesh.xyToIndex[x+2][y]] -
        signedDistance[mesh.xyToIndex[x+1][y]];
    v3 = signedDistance[mesh.xyToIndex[x+1][y]] -
        signedDistance[mesh.xyToIndex[x][y]];
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
        signedDistance[mesh.xyToIndex[x-1][y]];
    v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
        signedDistance[mesh.xyToIndex[x-2][y]];
}

double gradRight = sign * gradHJWENO(v1, v2, v3, v4, v5);

// Derivatives to left.

// Node on right-hand edge.
if (x == mesh.width)
{
    v1 = signedDistance[mesh.xyToIndex[x-3][y]] -
        signedDistance[mesh.xyToIndex[x-2][y]];
    v2 = signedDistance[mesh.xyToIndex[x-2][y]] -
        signedDistance[mesh.xyToIndex[x-1][y]];
    v3 = signedDistance[mesh.xyToIndex[x-1][y]] -
        signedDistance[mesh.xyToIndex[x][y]];
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
        signedDistance[mesh.xyToIndex[x-1][y]];

    // Approximate derivatives outside of domain.
    v4 = v3;
    v5 = v3;
}
else if (x == (mesh.width-1))
{
    v1 = signedDistance[mesh.xyToIndex[x-2][y]] -
        signedDistance[mesh.xyToIndex[x-3][y]];  
    v2 = signedDistance[mesh.xyToIndex[x-1][y]] -
        signedDistance[mesh.xyToIndex[x-2][y]];  
    v3 = signedDistance[mesh.xyToIndex[x][y]] -
        signedDistance[mesh.xyToIndex[x-1][y]];  
    v4 = signedDistance[mesh.xyToIndex[x+1][y]] -
        signedDistance[mesh.xyToIndex[x][y]];  

    // Approximate derivatives outside of domain.
    v5 = v4;
}

else if (x == 0)
{
    v5 = signedDistance[mesh.xyToIndex[2][y]] -
        signedDistance[mesh.xyToIndex[1][y]];  
    v4 = signedDistance[mesh.xyToIndex[1][y]] -
        signedDistance[mesh.xyToIndex[0][y]];  

    // Approximate derivatives outside of domain.
    v3 = v4;  
    v2 = v4;  
    v1 = v4;
}

else if (x == 1)
{
    v5 = signedDistance[mesh.xyToIndex[3][y]] -
        signedDistance[mesh.xyToIndex[2][y]];  
    v4 = signedDistance[mesh.xyToIndex[2][y]] -
        signedDistance[mesh.xyToIndex[1][y]];  
    v3 = signedDistance[mesh.xyToIndex[1][y]] -
        signedDistance[mesh.xyToIndex[0][y]];  

// Approximate derivatives outside of domain.
v2 = v3;
v1 = v3;
}

// Two nodes to right of left-hand edge.
else if (x == 2)
{
    v5 = signedDistance[mesh.xyToIndex[4][y]] -
         signedDistance[mesh.xyToIndex[3][y]];
    v4 = signedDistance[mesh.xyToIndex[3][y]] -
         signedDistance[mesh.xyToIndex[2][y]];
    v3 = signedDistance[mesh.xyToIndex[2][y]] -
         signedDistance[mesh.xyToIndex[1][y]];
    v2 = signedDistance[mesh.xyToIndex[1][y]] -
         signedDistance[mesh.xyToIndex[0][y]];

    // Approximate derivatives outside of domain.
    v1 = v2;
}

// Node lies in bulk.
else
{
    v1 = signedDistance[mesh.xyToIndex[x-2][y]] -
         signedDistance[mesh.xyToIndex[x-3][y]];
    v2 = signedDistance[mesh.xyToIndex[x-1][y]] -
         signedDistance[mesh.xyToIndex[x-2][y]];
    v3 = signedDistance[mesh.xyToIndex[x][y]] -
         signedDistance[mesh.xyToIndex[x-1][y]];
    v4 = signedDistance[mesh.xyToIndex[x+1][y]] -
         signedDistance[mesh.xyToIndex[x][y]];
    v5 = signedDistance[mesh.xyToIndex[x+2][y]] -
         signedDistance[mesh.xyToIndex[x+1][y]];
}

double gradLeft = sign * gradHJWENO(v1, v2, v3, v4, v5);

// Upward derivatives.

// Node on bottom edge.
if (y == 0)
{
    v1 = signedDistance[mesh.xyToIndex[x][3]] -
        signedDistance[mesh.xyToIndex[x][2]];  
    v2 = signedDistance[mesh.xyToIndex[x][2]] -
        signedDistance[mesh.xyToIndex[x][1]];  
    v3 = signedDistance[mesh.xyToIndex[x][1]] -
        signedDistance[mesh.xyToIndex[x][0]];  

    // Approximate derivatives outside of domain.
    v4 = v3;  
    v5 = v3;  
}

// One node above bottom edge.
else if (y == 1)
{
    v1 = signedDistance[mesh.xyToIndex[x][4]] -
        signedDistance[mesh.xyToIndex[x][3]];  
    v2 = signedDistance[mesh.xyToIndex[x][3]] -
        signedDistance[mesh.xyToIndex[x][2]];  
    v3 = signedDistance[mesh.xyToIndex[x][2]] -
        signedDistance[mesh.xyToIndex[x][1]];  
    v4 = signedDistance[mesh.xyToIndex[x][1]] -
        signedDistance[mesh.xyToIndex[x][0]];  

    // Approximate derivatives outside of domain.
    v5 = v4;  
}

// Node is on top edge.
else if (y == mesh.height)
{
    v5 = signedDistance[mesh.xyToIndex[x][y-1]] -
        signedDistance[mesh.xyToIndex[x][y-2]];  
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
        signedDistance[mesh.xyToIndex[x][y-1]];  

    // Approximate derivatives outside of domain.
    v3 = v4;  
    v2 = v4;  
}
v1 = v4;
}

// One node below top edge.
else if (y == (mesh.height - 1))
{
    v5 = signedDistance[mesh.xyToIndex[x][y-1]] -
        signedDistance[mesh.xyToIndex[x][y-2]];
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
        signedDistance[mesh.xyToIndex[x][y-1]];
    v3 = signedDistance[mesh.xyToIndex[x][y+1]] -
        signedDistance[mesh.xyToIndex[x][y]];  
    // Approximate derivatives outside of domain.
    v2 = v3;
    v1 = v3;
}

// Two nodes below top edge.
else if (y == (mesh.height - 2))
{
    v5 = signedDistance[mesh.xyToIndex[x][y-1]] -
        signedDistance[mesh.xyToIndex[x][y-2]];
    v4 = signedDistance[mesh.xyToIndex[x][y]] -
        signedDistance[mesh.xyToIndex[x][y-1]];
    v3 = signedDistance[mesh.xyToIndex[x][y+1]] -
        signedDistance[mesh.xyToIndex[x][y]];
    v2 = signedDistance[mesh.xyToIndex[x][y+2]] -
        signedDistance[mesh.xyToIndex[x][y+1]];
    // Approximate derivatives outside of domain.
    v1 = v2;
}

// Node lies in bulk.
else
{
    v1 = signedDistance[mesh.xyToIndex[x][y+3]] -
        signedDistance[mesh.xyToIndex[x][y+2]];
    v2 = signedDistance[mesh.xyToIndex[x][y+2]] -
        signedDistance[mesh.xyToIndex[x][y+1]];

v3 = signedDistance[mesh.xyToIndex[x][y+1]] - signedDistance[mesh.xyToIndex[x][y]];

v4 = signedDistance[mesh.xyToIndex[x][y]] - signedDistance[mesh.xyToIndex[x][y-1]];

v5 = signedDistance[mesh.xyToIndex[x][y-1]] - signedDistance[mesh.xyToIndex[x][y-2]];

}

double gradUp = sign * gradHJWENO(v1, v2, v3, v4, v5);

// Downward derivative.

// Node on top edge.
if (y == mesh.height)
{
   v1 = signedDistance[mesh.xyToIndex[x][y-2]] - signedDistance[mesh.xyToIndex[x][y-3]];
   v2 = signedDistance[mesh.xyToIndex[x][y-1]] - signedDistance[mesh.xyToIndex[x][y-2]];
   v3 = signedDistance[mesh.xyToIndex[x][y]] - signedDistance[mesh.xyToIndex[x][y-1]];
   
   // Approximate derivatives outside of domain.
   v4 = v3;
   v5 = v3;
}

// One node below top edge.
else if (y == (mesh.height - 1))
{
   v1 = signedDistance[mesh.xyToIndex[x][y-2]] - signedDistance[mesh.xyToIndex[x][y-3]];
   v2 = signedDistance[mesh.xyToIndex[x][y-1]] - signedDistance[mesh.xyToIndex[x][y-2]];
   v3 = signedDistance[mesh.xyToIndex[x][y]] - signedDistance[mesh.xyToIndex[x][y-1]];
   v4 = signedDistance[mesh.xyToIndex[x][y+1]] - signedDistance[mesh.xyToIndex[x][y]];
   
   // Approximate derivatives outside of domain.
   v5 = v4;
else if (y == 0)
{
    v5 = signedDistance[mesh.xyToIndex[x][2]] -
        signedDistance[mesh.xyToIndex[x][1]];
    v4 = signedDistance[mesh.xyToIndex[x][1]] -
        signedDistance[mesh.xyToIndex[x][0]];

    // Approximate derivatives outside of domain.
    v3 = v4;
    v2 = v4;
    v1 = v4;
}

else if (y == 1)
{
    v5 = signedDistance[mesh.xyToIndex[x][3]] -
        signedDistance[mesh.xyToIndex[x][2]];
    v4 = signedDistance[mesh.xyToIndex[x][2]] -
        signedDistance[mesh.xyToIndex[x][1]];
    v3 = signedDistance[mesh.xyToIndex[x][1]] -
        signedDistance[mesh.xyToIndex[x][0]];

    // Approximate derivatives outside of domain.
    v2 = v3;
    v1 = v3;
}

else if (y == 2)
{
    v5 = signedDistance[mesh.xyToIndex[x][4]] -
        signedDistance[mesh.xyToIndex[x][3]];
    v4 = signedDistance[mesh.xyToIndex[x][3]] -
        signedDistance[mesh.xyToIndex[x][2]];
    v3 = signedDistance[mesh.xyToIndex[x][2]] -
        signedDistance[mesh.xyToIndex[x][1]];
    v2 = signedDistance[mesh.xyToIndex[x][1]] -
signedDistance[mesh.xyToIndex[x][0]];

    // Approximate derivatives outside of domain.
    v1 = v2;
}

    // Node lies in bulk.
else
{
    v1 = signedDistance[mesh.xyToIndex[x][y-2]] -
        signedDistance[mesh.xyToIndex[x][y-3]];
    v2 = signedDistance[mesh.xyToIndex[x][y-1]] -
        signedDistance[mesh.xyToIndex[x][y-2]];
    v3 = signedDistance[mesh.xyToIndex[x][y]] -
        signedDistance[mesh.xyToIndex[x][y-1]];
    v4 = signedDistance[mesh.xyToIndex[x][y+1]] -
        signedDistance[mesh.xyToIndex[x][y]];
    v5 = signedDistance[mesh.xyToIndex[x][y+2]] -
        signedDistance[mesh.xyToIndex[x][y+1]];
}

double gradDown = sign * gradHJWENO(v1, v2, v3, v4, v5);

    // Compute gradient using upwind scheme.
if (gradDown > 0) grad += gradDown * gradDown;
if (gradLeft > 0) grad += gradLeft * gradLeft;
if (gradUp < 0)  grad += gradUp * gradUp;
if (gradRight < 0) grad += gradRight * gradRight;

    grad = sqrt(grad);
}

    // Return gradient.
return grad;
}

4.3.4 Fast Marching Method

The level set advection is done by using the Fast Marching Method and is implemented according to the following code snippet:
void FastMarchingMethod::march(const std::vector<double>& signedDistance_,
std::vector<double>& velocity_)
{
    // Extend boundary velocities to all nodes within the narrow band region.
    // Note that this method assumes that boundary point velocities have already been
    // mapped to the level set nodes using inverse squared distance interpolation, or similar.
    /*
    std::cout << velocity_.size() << std::endl;
    std::cout << signedDistance_.size() << std::endl;
    std::cout << velocity_.size() == signedDistance_.size() << std::endl;
    */
    for (size_t i = 0; i < velocity_.size(); ++i) {
        // Extend boundary velocities to all nodes within the narrow band region.
        // Note that this method assumes that boundary point velocities have already been
        // mapped to the level set nodes using inverse squared distance interpolation, or similar.
    }
}
signedDistance = &signedDistance_;  
velocity = &velocity_;  
isVelocity = true;  

// Initialise the set of frozen boundary nodes.  
initialiseFrozen();  

// Initialise the heap data structure.  
initialiseHeap();  

// Initialise the set of trial nodes adjacent to the boundary.  
initialiseTrial();  

// Find the fast marching solution.  
solve();  

// Restore the original signed distance function. Only update velocities.  
(*signedDistance) = signedDistanceCopy;  
}