Programmer Manual for OpenLSTO v1.0:

Open Source Level Set Topology Optimization

M2DO Lab[1,2]
[1]Cardiff University
[2]University of California, San Diego
August 2018

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.

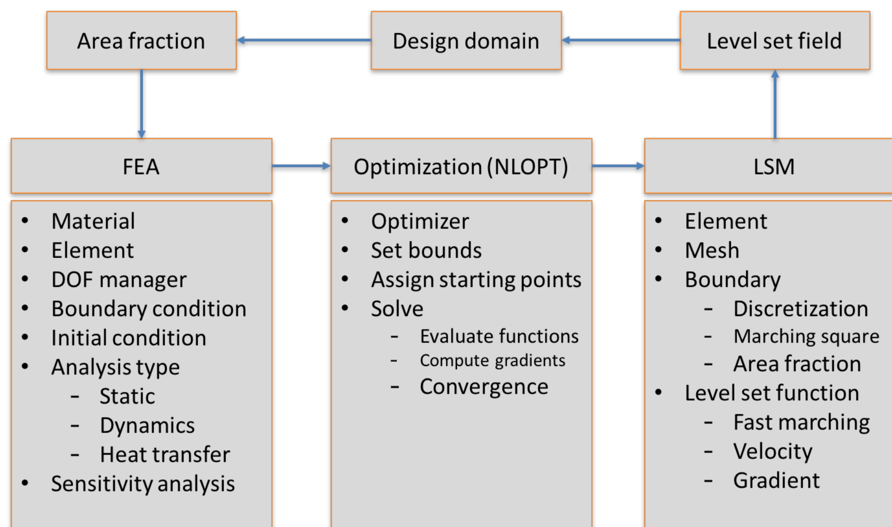

Figure 2.1: General structure of OpenLSTO framework.

An example of key elements for a typical main file scope for implementing OpenLSTO is given below:

```
#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:

```
1  #ifndef M2DO_FEA_MODULE_H
2  #define M2DO_FEA_MODULE_H
3
4  ...
5
6  using namespace std ;
7
8  #include "quadrature.h"
9  #include "linear_shape_function.h"
10 #include "node.h"
11 #include "solid_element.h"
12 #include "solid_material.h"
13 #include "mesh.h"
```

```
14 #include "boundary_conditions.h"
15 #include "stationary_study.h"
16 #include "sensitivity.h"
17
18 #endif
```

An instance of a finite element model is declared as follows:

```
1   // FEA mesh object for 2D analysis:
2   FEA::Mesh fea_mesh(2) ;
3
4   // FEA mesh object for 3D analysis:
5   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:

```
1   double   E = 1.0 ; // Young's Modulus
2   double  nu = 0.3 ; // Poisson's ratio
3   double rho = 1.0 ; // Density
4   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:

```
1 SolidMaterial :: SolidMaterial (int spacedim, double E, double nu, double
    rho, double h) : spacedim (spacedim), E (E), nu (nu), rho (rho), h (h)
    {
2
3    if (spacedim == 2) {
4
5      Matrix<double,-1,-1> A(4,4);
```

```
6
7      A.data = {{1, 0,  0,  0},
8         {0, 0.5, 0.5, 0},
9         {0, 0.5, 0.5, 0},
10        {0,  0,   0,  1}} ;
11
12     // Voight matrix:
13     Matrix<double,-1,-1> V(4,4);
14
15     V.data ={{ 1,  0,   0, -0.5 },
16           {0, 1.5,  0,    0},
17           {0,   0, 1.5,   0},
18        {-0.5,  0,   0,    1}} ;
19
20     // Note: This is the plane stress formulation!
21     Matrix<double,-1,-1> D(4,4);
22     D.data = {{1,    0,         0,    nu},
23        {0, (1-nu)/2, (1-nu)/2, 0},
24        {0, (1-nu)/2, (1-nu)/2, 0},
25        {nu,    0,       0,    1 }};
26
27     D *= E / (1-pow(nu,2)) ;
28
29     C = (D.dot(A));
30     C *= h;
31
32   }
33
34   else if (spacedim == 3) {
35
36     Matrix<double,-1,-1> A(9,9);
37
38     A.data = {{ 1, 0,   0,   0, 0,   0,   0,  0,  0},
39        {0, 0.5,  0, 0.5, 0,  0,    0,  0,  0},
40        {0,   0, 0.5,  0, 0,   0, 0.5,  0,  0},
41        {0, 0.5,  0, 0.5, 0,  0,    0,  0,  0},
42        {0,   0,   0,  0, 1,  0,    0,  0,  0},
43        {0,   0,   0,  0, 0, 0.5,  0, 0.5, 0},
44        {0,   0, 0.5,  0, 0,   0, 0.5,  0,  0},
45        {0,   0,   0,  0, 0, 0.5,  0, 0.5, 0},
46        {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},
              {0, 0, 0, (1-2*nu), 0, 0, 0, 0, 0},
             {nu, 0, 0, 0, (1-nu), 0, 0, 0, nu},
              {0, 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},
             {nu, 0, 0, 0, nu, 0, 0, 0, (1-nu) }};

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

    C = D.dot(A) ;

  }

}
```

3.2 Design Domain's Geometry and Finite Element Discretization

Typically, a rectangular design domain for 2D problem or a rectangular cuboid for 3D are defined due to the utilisation of fixed grid finite element method. For instance, a rectangle is declared here with giving coordinates of its four corners using `fea_box`:

```
// fea_box contains the (x,y) coordinates of 4 corner points of
    rectangle containing the mesh:
Matrix<double, -1, -1> fea_box (4, 2) ;
fea_box.data = {{ 0,   0}, {nelx,   0},  {nelx, nely}, { 0, nely}} ;
```

3.2.1 Element

Current version of the code supports both 2D and 3D plane stress or plane strain problems. For 2D meshes, each node of the elements has degrees of freedom of x- and y-direction displacements. For 3D meshes, the solid element implemented has

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

```
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 ;

```

```
34        for (int l = 0 ; l < spacedim ; ++l) {
35
36           eta[l]  = quadrature.eta[eta_count[l]] ;
37           w      *= quadrature.w[eta_count[l]]  ;
38
39        }
40
41        J_mat = J (eta) ;
42        J_inv = J_mat.inverse() ;
43
44        /*
45           Build the B matrix at this gauss point:
46        */
47
48        for (int j = 0 ; j < spacedim * pow(2, spacedim) ; ++j) {
49
50           shape_grad_j     =
                J_inv.dot(linear_shape_function.GetShapeFunctionGradientsVector(shape_j,
                eta)) ;
51           shape_grad_j_full =
                linear_shape_function.GetShapeFunctionGradientsFullVector(shape_grad_j,
                dim_j) ;
52
53           // hac210 note: col.
54           // B_mat.col(j)     = shape_grad_j_full ;
55           for (int kk = 0; kk < B_mat.rows(); kk ++){
56              B_mat(kk,j)     = shape_grad_j_full(kk) ;
57           }
58
59           if (dim_j < spacedim-1) {
60              dim_j = dim_j + 1 ;
61           }
62
63           else {
64              dim_j = 0 ;
65              shape_j = shape_j + 1 ;
66           }
67
68        } // for j (columns in B).
69
70        eta_count = quadrature.UpdateEtaCounter(eta_count) ;
```

```
71
72      /*
73         Add to the K matrix:
74      */
75
76      // K_mat += B_mat.transpose() * C * B_mat * w * J_mat.determinant() ;
77
78      // hac210 note: * is needed for scalar constant multiplication
79      Matrix<double,-1,-1> Bt, K_tmp;
80      Bt = B_mat.transpose();
81      K_tmp = (Bt.dot(C.dot(B_mat)));
82      K_tmp *= (w * J_mat.determinant());
83      for(int ii = 0; ii < K_tmp.rows(); ii++) for(int jj = 0; jj <
           K_tmp.cols(); jj++) K_mat(ii,jj)+= K_tmp(ii,jj);
84
85
86    } // for k (gauss points).
87    //for(int ii = 0; ii < K_mat.rows(); ii++) for(int jj = 0; jj <
         K_mat.cols(); jj++) cout << K_mat(ii,jj) << endl;
88    return K_mat ;
89
90  }
```

The corresponding strain-displacement matrix, **B**, is accomplished as:

```
1   Matrix<double,-1,-1> SolidElement :: B (vector<double> & eta) {
2
3     Vector<double,-1> shape_grad_j, shape_grad_j_full ;
4
5     /*
6        grad(u(x)) = [B] * {u}
7        So we aim to find B here.
8     */
9
10    Matrix<double,-1,-1> B_mat(spacedim * spacedim, pow(2, spacedim) *
         spacedim) ;
11    B_mat.fill(0.0);
12
13    int shape_j = 0, dim_j = 0 ;
14
15    Matrix<double,-1,-1> J_mat = J (eta) ;
16    Matrix<double,-1,-1> J_inv = J_mat.inverse() ;
```

```
17
18    // Build the B matrix at the given eta point:
19    for (int j = 0 ; j < spacedim * pow(2, spacedim) ; ++j) {
20
21       shape_grad_j     =
            J_inv.dot(linear_shape_function.GetShapeFunctionGradientsVector(shape_j,
            eta)) ;
22       shape_grad_j_full =
            linear_shape_function.GetShapeFunctionGradientsFullVector(shape_grad_j,
            dim_j) ;
23
24       // hac210 note: col.
25       // B_mat.col(j)     = shape_grad_j_full ;
26       for (int kk = 0; kk < B_mat.rows(); kk ++){
27          B_mat(kk,j)     = shape_grad_j_full(kk) ;
28       }
29
30       if (dim_j < spacedim-1) {
31          dim_j = dim_j + 1 ;
32       }
33
34       else {
35          dim_j = 0 ;
36          shape_j = shape_j + 1 ;
37       }
38
39    } // for j (columns in B).
40
41    return B_mat ;
42
43 }
```

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:

```
1  // Element Gauss integration order:
2  int element_order = 2 ;
```

```
3   // Create structured mesh and assign degrees of freedom:
4   fea_mesh.MeshSolidHyperRectangle ({nelx, nely}, fea_box, element_order,
       false) ;
5   fea_mesh.is_structured = true ;
6   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 conner 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:

```
1  void Mesh :: MeshSolidHyperRectangle (vector<int> nel,
       Matrix<double,-1,-1> mesh_box, int element_order, bool time_it) {
2
3     auto t_start = chrono::high_resolution_clock::now() ;
4
5     if (time_it) {
6
7        cout << "\nMeshing solid hyper-rectangle ... " << flush ;
8
9     }
10
11    /*
12       First mesh the natural hyper-rectangle. We can utilise a sequence
13       similar to that in quadrature.UpdateEtaCounter() for this.
14    */
15
16    int num_nodes = 1, num_elements = 1 ;
17
18    for (int i = 0 ; i < nel.size() ; ++i) {
19
20       num_nodes    *= nel[i]+1 ;
21       num_elements *= nel[i] ;
22
23    }
24
25    nodes.reserve (num_nodes) ;
```

```
26    solid_elements.reserve (num_elements) ;
27
28    /*
29      Each of these linear elements has pow (2, spacedim) nodes;
30      each node has dim degrees of freedom. Using this we can
31      calculate num_entries (needed for study step).
32    */
33
34    // num_entries += num_elements * pow((pow(2, spacedim) * dim), 2) ;
35    // n_dof       = num_nodes * dim ;
36
37    vector<int> eta_count (spacedim, 0) ;
38
39    for (int i = 0 ; i < num_nodes ; ++i) {
40
41      Node node (spacedim) ;
42      node.id = i ;
43
44      for (int l = 0 ; l < spacedim ; ++l) {
45
46        node.coordinates[l] = -1 + eta_count[l] * (2.0 / nel[l]) ;
47
48      }
49
50      nodes.push_back(node) ;
51
52      // Update eta_count:
53
54      eta_count[0] += 1 ;
55
56      if (eta_count[0] > nel[0]) {
57
58        eta_count[0] = 0 ;
59
60        for (int l = 1 ; l < spacedim ; ++l) {
61
62          eta_count[l] += 1 ;
63
64          if (eta_count[l] <= nel[l]) {
65            break ;
66          }
```

```
67
68            else {
69               eta_count[l] = 0 ;
70            }
71
72         }
73
74      }
75
76   }
77
78   /*
79      We create the elements on the natural hyper_rectangle,
80      as it is easy to visualize. The node locations will change
81      below, but that doesn't affect the elements; they need
82      only know which nodes belong to them.
83   */
84
85   LinearShapeFunction linear_shape_function (spacedim, 1) ;
86
87   vector<int> node_ids (pow(2, spacedim), 0) ;
88   vector<int> nel_count (spacedim, 0) ;
89   vector<int> nel_mult (spacedim, 1) ;
90   vector<double> eta (spacedim, 0) ;
91   int eta_int, start_id ;
92
93   for (int l = 1 ; l < spacedim ; ++l) {
94
95      nel_mult[l] *= nel_mult[l-1] * (nel[l-1]+1) ;
96
97   }
98
99   SolidElement element (spacedim, element_order, *this) ;
100
101  for (int i = 0 ; i < num_elements ; ++i) {
102
103     start_id = 0 ;
104
105     for (int l = 0 ; l < sp here. acedim ; ++l) {
106
107        start_id += nel_count[l] * nel_mult[l] ;
```

```
108
109     }
110
111   // This just fills the node_ids vector with start_id:
112     fill(node_ids.begin(), node_ids.end(), start_id) ;
113
114     for (int j = 0 ; j < pow(2, spacedim) ; ++j) {
115
116       eta = linear_shape_function.GetEta(j) ;
117
118       // Change the -1 values to zeros. Also, eta
119       // comes as doubles so change to int, then
120       // multiply by nel_mult:
121
122       for (int k = 0 ; k < spacedim ; ++k) {
123
124         eta_int = (eta[k] < 0) ? 0 : 1 ;
125         node_ids[j] += eta_int * nel_mult[k] ;
126
127       }
128
129       element.node_ids[j] = node_ids[j] ;
130
131     }
132
133     // Add the element to the mesh:
134     solid_elements.push_back(element) ;
135
136     // Update nel_count:
137
138     nel_count[0] += 1 ;
139
140     if (nel_count[0] > nel[0]-1) {
141
142       nel_count[0] = 0 ;
143
144       for (int l = 1 ; l < spacedim ; ++l) {
145
146         nel_count[l] += 1 ;
147
148         if (nel_count[l] < nel[l]) {
```

```
149               break ;
150            }
151
152            else {
153               nel_count[l] = 0 ;
154            }
155
156         }
157
158      }
159
160   }
161
162   /*
163      Now we deform the natural mesh to conform to the physical
164      geometry using linear shape functions:
165
166      x = sum(N_i * x_i) etc. where in this case x_i are the
167      coordinates of the corners of the hyper_rectangle to mesh.
168   */
169
170   Vector<double,-1> shape_value_vec ;
171
172   for (int i = 0 ; i < nodes.size() ; ++i) {
173
174      // Shape function values at natural coordinate:
175      shape_value_vec =
          linear_shape_function.GetShapeFunctionValuesVector(nodes[i].coordinates)
          ;
176
177      for (int j = 0 ; j < spacedim ; ++j) {
178
179         // hac210 note: cols
180         nodes[i].coordinates[j] = 0.0;
181
182         for (int pp = 0; pp < mesh_box.rows(); pp ++){
183            nodes[i].coordinates[j] += mesh_box(pp,j)*shape_value_vec(pp) ;
184         }
185      }
186
187   }
```

```
188
189     auto t_end = chrono::high_resolution_clock::now() ;
190
191     if (time_it) {
192
193         cout << "Done. Time elapsed = " <<
              chrono::duration<double>(t_end-t_start).count() << "\n" << flush ;
194
195     }
196
197 }
```

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:

```
1  void Mesh :: AssignDof () {
2
3     n_dof = 0 ;
4
5     /*
6        Solid elements: these have spacedim displacement dofs per node,
7        and there are pow(2, spacedim) nodes per element.
8     */
9
10    for (auto && element : solid_elements) {
11
12       element.dof = vector<int> (spacedim * pow (2, spacedim), -1) ;
13
14       for (int i = 0 ; i < element.node_ids.size() ; ++i) {
15
16          auto && node = nodes[element.node_ids[i]] ;
17
18          /*
19             Check if this node has already assigned a dof number;
20             If not, assign one:
21          */
22
23          for (int j = 0 ; j < spacedim ; ++j) {
24
25             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

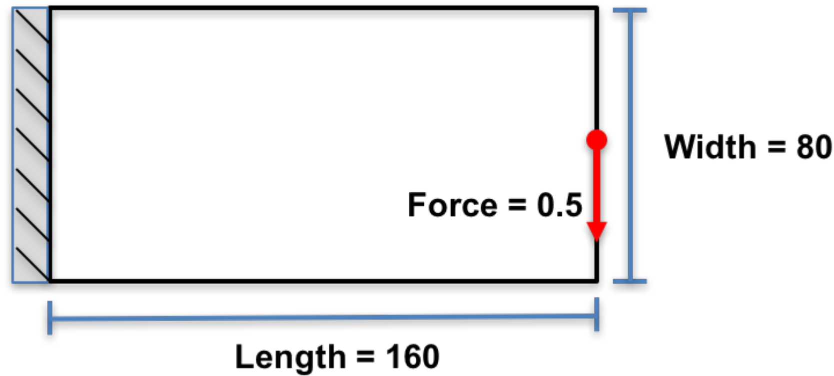

Figure 3.1: Configuration of the cantilever beam subject to a point load.

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:

```
// 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:

```
 // 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:

$$\mathbf{K_e} = x_i \mathbf{K_e^0} \tag{3.1}$$

Here, $\mathbf{K_e}$ is the elemental stiffness matrix, x_i is the element area fraction of the solid portion determined by the level set and $\mathbf{K_e^0}$ 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 $\mathbf{K}(i, j) = v$, then the sparse representation for this value would be:

```
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

$$\alpha_k = \frac{r_k^T r_k}{p_k^T K p_k}$$

$$u_{k+1} = u_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k \mathbf{K} p_k$$

$$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k$$

$$k = k + 1$$

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:

```
1 void StationaryStudy ::mat_vec_mult( vector<Triplet_Sparse> &K,
    vector<double> &v_in, vector<double> &v_out ){
2   #pragma omp parallel for
3   for(int i = 0; i < K.size(); i++) {
4       v_out[K[i].row] += K[i].val*v_in[K[i].col];
5   }
6 }
```

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:

```
#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"
```

```
    #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:

```
// 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:

```
Mesh::Mesh(unsigned int width_,
           unsigned int height_,
           bool isPeriodic_) :

           width(width_),
           height(height_),
           nElements(width*height),
           nNodes((1+width)*(1+height)),
           isPeriodic(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:

```
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:

```
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++)
```

```
    {
        // Loop over all holes.
        for (unsigned int j=0;j<holes.size();j++)
        {
            // Work out x and y distance of the node from the hole centre.
            double dx = holes[j].coord.x - mesh.nodes[i].coord.x;
            double dy = holes[j].coord.y - mesh.nodes[i].coord.y;

            // Work out distance (Pythag).
            double dist = sqrt(dx*dx + dy*dy);

            // Signed distance from the hole surface.
            dist -= holes[j].r;

            // If distance is less than current value, then update.
            if (dist < signedDistance[i])
                signedDistance[i] = dist;
        }
    }
}
```

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:

```
// 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:

```
//! 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];

```

```
28          // Node matches status.
29          if (levelSet.mesh.nodes[node].status & status)
30          {
31              // Add coordinates to vertex array.
32              vertices[nVertices].x = levelSet.mesh.nodes[node].coord.x;
33              vertices[nVertices].y = levelSet.mesh.nodes[node].coord.y;
34
35              // Increment number of vertices.
36              nVertices++;
37          }
38
39          // Node is on the boundary.
40          else if (levelSet.mesh.nodes[node].status & NodeStatus::BOUNDARY)
41          {
42              // Next node.
43              unsigned int n1 = (i == 3) ? 0 : (i + 1);
44              n1 = element.nodes[n1];
45
46              // Previous node.
47              unsigned int n2 = (i == 0) ? 3 : (i - 1);
48              n2 = element.nodes[n2];
49
50              // Check that node isn't part of a boundary segment, i.e. both
                    of its
51              // neighbours are inside the structure.
52              if ((levelSet.mesh.nodes[n1].status & NodeStatus::INSIDE) &&
53                  (levelSet.mesh.nodes[n2].status & NodeStatus::INSIDE))
54              {
55                  // Add coordinates to vertex array.
56                  vertices[nVertices].x = levelSet.mesh.nodes[node].coord.x;
57                  vertices[nVertices].y = levelSet.mesh.nodes[node].coord.y;
58
59                  // Increment number of vertices.
60                  nVertices++;
61              }
62          }
63      }
64
65      // Add boundary segment start and end points.
66      for (unsigned int i=0;i<element.nBoundarySegments;i++)
67      {
```

```
68          // Segment index.
69          unsigned int segment = element.boundarySegments[i];
70
71          // Add start point coordinates to vertices array.
72          vertices[nVertices].x = points[segments[segment].start].coord.x;
73          vertices[nVertices].y = points[segments[segment].start].coord.y;
74
75          // Increment number of vertices.
76          nVertices++;
77
78          // Add end point coordinates to vertices array.
79          vertices[nVertices].x = points[segments[segment].end].coord.x;
80          vertices[nVertices].y = points[segments[segment].end].coord.y;
81
82          // Increment number of vertices.
83          nVertices++;
84      }
85
86      // Return area of the polygon.
87      if (element.status & ElementStatus::CENTRE_OUTSIDE)
88          return (1.0 - polygonArea(vertices, nVertices, element.coord));
89      else
90          return polygonArea(vertices, nVertices, element.coord);
91  }
92
93  //! Calculate the material area fraction in each element.
94  /*! \return
95          The total element area fraction.
96   */
97  double Boundary::computeAreaFractions()
98  {
99      // Zero the total area fraction.
100     area = 0;
101
102     for (unsigned int i=0;i<levelSet.mesh.nElements;i++)
103     {
104         // Element is inside structure.
105         if (levelSet.mesh.elements[i].status & ElementStatus::INSIDE)
106             levelSet.mesh.elements[i].area = 1.0;
107
108         // 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:

$$\begin{aligned} \text{Minimize} \quad & C = C_0 + \sum_i^N C_i^f z_i \\ \text{subject to} \quad & A = A_0 + \sum_i^N C_i^g z_i \leq A_{max} \\ & z_{min} \leq z_i \leq z_{max} \\ & z_i = \lambda_f s_f^i + \lambda_g s_g^i; \quad i = 1...N \end{aligned} \tag{4.1}$$

where

z_i is the displacement of the ith 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 ith 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 ith boundary point, multiplied by the segment length (l_i);

z_{max} and z_{min} are the maximum and minimum allowed displacement for the boundary point, respectively;

λ_f is the Lagrangian multiplier associated with the objective function;

λ_g is the Lagrangian multiplier associated with the constraint (area).

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

4.3.2 Velocity

Having obtained λ_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:

$$\nabla\phi \cdot \nabla V = 0 \tag{4.2}$$

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:

$$\frac{\partial\phi}{\partial t} + V|\nabla\phi| = 0. \tag{4.3}$$

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

```
1  //! Extend boundary point velocities to the level set nodes.
2  /*! \param boundaryPoints
3          A reference to a vector of boundary points.
4   */
5  void LevelSet::computeVelocities(const std::vector<BoundaryPoint>&
       boundaryPoints)
6  {
7      // Initialise velocity (map boundary points to boundary nodes).
8      initialiseVelocities(boundaryPoints);
9
10     // Initialise fast marching method object.
11     FastMarchingMethod fmm(mesh, false);
12
13     // Reinitialise the signed distance function.
14     fmm.march(signedDistance, velocity);
15 }
16
17 //! Initialise velocities for boundary nodes.
18 /*! \param boundaryPoints
19         A reference to a vector of boundary points.
20  */
21  void LevelSet::initialiseVelocities(const std::vector<BoundaryPoint>&
        boundaryPoints)
22 {
23     // Map boundary point velocities to nodes of the level set domain
24     // using inverse squared distance interpolation.
```

```
25
26      // Whether the velocity at a node has been set.
27      bool isSet[mesh.nNodes];
28
29      // Weighting factor for each node.
30      double weight[mesh.nNodes];
31
32      // Initialise arrays.
33      for (unsigned int i=0;i<mesh.nNodes;i++)
34      {
35          isSet[i] = false;
36          weight[i] = 0;
37          velocity[i] = 0;
38      }
39
40      // Loop over all boundary points.
41      for (unsigned int i=0;i<boundaryPoints.size();i++)
42      {
43          // Find the closest node.
44          unsigned int node = mesh.getClosestNode(boundaryPoints[i].coord);
45
46          // Distance from the boundary point to the node.
47          double dx = mesh.nodes[node].coord.x - boundaryPoints[i].coord.x;
48          double dy = mesh.nodes[node].coord.y - boundaryPoints[i].coord.y;
49
50          // Squared distance.
51          double rSqd = dx*dx + dy*dy;
52
53          // If boundary point lies exactly on the node, then set velocity
54          // to that of the boundary point.
55          if (rSqd < 1e-6)
56          {
57              velocity[node] = boundaryPoints[i].velocity;
58              weight[node] = 1.0;
59              isSet[node] = true;
60          }
61          else
62          {
63              // Update velocity estimate if not already set.
64              if (!isSet[node])
65              {
```

```
66                  velocity[node] += boundaryPoints[i].velocity / rSqd;
67                  weight[node] += 1.0 / rSqd;
68              }
69          }
70
71          // Loop over all neighbours of the node.
72          for (unsigned int j=0;j<4;j++)
73          {
74              // Index of the neighbouring node.
75              unsigned int neighbour = mesh.nodes[node].neighbours[j];
76
77              // Make sure neighbour is in bounds.
78              if (neighbour < mesh.nNodes)
79              {
80                  // Distance from the boundary point to the node.
81                  double dx = mesh.nodes[neighbour].coord.x -
                        boundaryPoints[i].coord.x;
82                  double dy = mesh.nodes[neighbour].coord.y -
                        boundaryPoints[i].coord.y;
83
84                  // Squared distance.
85                  double rSqd = dx*dx + dy*dy;
86
87                  // If boundary point lies exactly on the node, then set
                        velocity
88                  // to that of the boundary point.
89                  if (rSqd < 1e-6)
90                  {
91                      velocity[neighbour] = boundaryPoints[i].velocity;
92                      weight[neighbour] = 1.0;
93                      isSet[neighbour] = true;
94                  }
95                  else if (rSqd <= 1.0)
96                  {
97                      // Update velocity estimate if not already set.
98                      if (!isSet[neighbour])
99                      {
100                         velocity[neighbour] += boundaryPoints[i].velocity /
                                rSqd;
101                         weight[neighbour] += 1.0 / rSqd;
102                     }
```

```
                }
            }
        }
    }

    // Compute interpolated velocity.
    for (unsigned int i=0;i<nNarrowBand;i++)
    {
        unsigned int node = narrowBand[i];
        if (velocity[node]) velocity[node] /= weight[node];
    }
}
```

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:

```
//! Compute the modulus of the gradient of the signed distance function at
    a node.
/*! \param node
        The node index.

    \return
        The gradient at the node.
 */
 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;
```

```
22
23      // Node is on the left edge.
24      if (x == 0)
25      {
26          // Node is at bottom left corner.
27          if (y == 0)
28          {
29              // If signed distance at nodes to right and above is the same,
                    then use
30              // the diagonal node for computing the gradient.
31              if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
                    1e-6) &&
32                  (std::abs(signedDistance[mesh.xyToIndex[x][y+1]] - lsf) <
                        1e-6))
33              {
34                  // Calculate signed distance to diagonal node.
35                  grad = std::abs(lsf -
                        signedDistance[mesh.xyToIndex[x+1][y+1]]);
36                  grad *= sqrt(2.0);
37                  isGradient = true;
38              }
39          }
40
41          // Node is at top left corner.
42          else if (y == mesh.height)
43          {
44              // If signed distance at nodes to right and below is the same,
                    then use
45              // the diagonal node for computing the gradient.
46              if ((std::abs(signedDistance[mesh.xyToIndex[x+1][y]] - lsf) <
                    1e-6) &&
47                  (std::abs(signedDistance[mesh.xyToIndex[x][y-1]] - lsf) <
                        1e-6))
48              {
49                  // Calculate signed distance to diagonal node.
50                  grad = std::abs(lsf -
                        signedDistance[mesh.xyToIndex[x+1][y-1]]);
51                  grad *= sqrt(2.0);
52                  isGradient = true;
53              }
54          }
```

```
55      }
56
57      // Node is on the right edge.
58      else if (x == mesh.width)
59      {
60          // Node is at bottom right corner.
61          if (y == 0)
62          {
63              // If signed distance at nodes to left and above is the same,
                    then use
64              // the diagonal node for computing the gradient.
65              if ((std::abs(signedDistance[mesh.xyToIndex[x-1][y]] - lsf) <
                    1e-6) &&
66                  (std::abs(signedDistance[mesh.xyToIndex[x][y+1]] - lsf) <
                        1e-6))
67              {
68                  // Calculate signed distance to diagonal node.
69                  grad = std::abs(lsf -
                        signedDistance[mesh.xyToIndex[x-1][y+1]]);
70                  grad *= sqrt(2.0);
71                  isGradient = true;
72              }
73          }
74
75          // Node is at top right corner.
76          else if (y == mesh.height)
77          {
78              // If signed distance at nodes to left and below is the same,
                    then use
79              // the diagonal node for computing the gradient.
80              if ((std::abs(signedDistance[mesh.xyToIndex[x-1][y]] - lsf) <
                    1e-6) &&
81                  (std::abs(signedDistance[mesh.xyToIndex[x][y-1]] - lsf) <
                        1e-6))
82              {
83                  // Calculate signed distance to diagonal node.
84                  grad = std::abs(lsf -
                        signedDistance[mesh.xyToIndex[x-1][y-1]]);
85                  grad *= sqrt(2.0);
86                  isGradient = true;
87              }
```

```
88          }
89      }
90
91      // Gradient hasn't already been calculated.
92      if (!isGradient)
93      {
94          // Stencil values for the WENO approximation.
95          double v1, v2, v3, v4, v5;
96
97          // Upwind direction.
98          int sign = velocity[node] < 0 ? -1 : 1;
99
100         // Derivatives to right.
101
102         // Node on left-hand edge.
103         if (x == 0)
104         {
105             v1 = signedDistance[mesh.xyToIndex[3][y]] -
                    signedDistance[mesh.xyToIndex[2][y]];
106             v2 = signedDistance[mesh.xyToIndex[2][y]] -
                    signedDistance[mesh.xyToIndex[1][y]];
107             v3 = signedDistance[mesh.xyToIndex[1][y]] -
                    signedDistance[mesh.xyToIndex[0][y]];
108
109             // Approximate derivatives outside of domain.
110             v4 = v3;
111             v5 = v3;
112         }
113
114         // One node to right of left-hand edge.
115         else if (x == 1)
116         {
117             v1 = signedDistance[mesh.xyToIndex[4][y]] -
                    signedDistance[mesh.xyToIndex[3][y]];
118             v2 = signedDistance[mesh.xyToIndex[3][y]] -
                    signedDistance[mesh.xyToIndex[2][y]];
119             v3 = signedDistance[mesh.xyToIndex[2][y]] -
                    signedDistance[mesh.xyToIndex[1][y]];
120             v4 = signedDistance[mesh.xyToIndex[1][y]] -
                    signedDistance[mesh.xyToIndex[0][y]];
121
```

```
122             // Approximate derivatives outside of domain.
123             v5 = v4;
124         }
125 
126         // Node on right-hand edge.
127         else if (x == mesh.width)
128         {
129             v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
                    signedDistance[mesh.xyToIndex[x-2][y]];
130             v4 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x-1][y]];
131 
132             // Approximate derivatives outside of domain.
133             v3 = v4;
134             v2 = v4;
135             v1 = v4;
136         }
137 
138         // One node to left of right-hand edge.
139         else if (x == (mesh.width - 1))
140         {
141             v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
                    signedDistance[mesh.xyToIndex[x-2][y]];
142             v4 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x-1][y]];
143             v3 = signedDistance[mesh.xyToIndex[x+1][y]] -
                    signedDistance[mesh.xyToIndex[x][y]];
144 
145             // Approximate derivatives outside of domain.
146             v2 = v3;
147             v1 = v3;
148         }
149 
150         // Two nodes to left of right-hand edge.
151         else if (x == (mesh.width - 2))
152         {
153             v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
                    signedDistance[mesh.xyToIndex[x-2][y]];
154             v4 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x-1][y]];
155             v3 = signedDistance[mesh.xyToIndex[x+1][y]] -
```

```
                signedDistance[mesh.xyToIndex[x][y]];
156             v2 = signedDistance[mesh.xyToIndex[x+2][y]] -
                signedDistance[mesh.xyToIndex[x+1][y]];
157
158             // Approximate derivatives outside of domain.
159             v1 = v2;
160         }
161
162         // Node lies in bulk.
163         else
164         {
165             v1 = signedDistance[mesh.xyToIndex[x+3][y]] -
                signedDistance[mesh.xyToIndex[x+2][y]];
166             v2 = signedDistance[mesh.xyToIndex[x+2][y]] -
                signedDistance[mesh.xyToIndex[x+1][y]];
167             v3 = signedDistance[mesh.xyToIndex[x+1][y]] -
                signedDistance[mesh.xyToIndex[x][y]];
168             v4 = signedDistance[mesh.xyToIndex[x][y]] -
                signedDistance[mesh.xyToIndex[x-1][y]];
169             v5 = signedDistance[mesh.xyToIndex[x-1][y]] -
                signedDistance[mesh.xyToIndex[x-2][y]];
170         }
171
172         double gradRight = sign * gradHJWENO(v1, v2, v3, v4, v5);
173
174         // Derivatives to left.
175
176         // Node on right-hand edge.
177         if (x == mesh.width)
178         {
179             v1 = signedDistance[mesh.xyToIndex[x-2][y]] -
                signedDistance[mesh.xyToIndex[x-3][y]];
180             v2 = signedDistance[mesh.xyToIndex[x-1][y]] -
                signedDistance[mesh.xyToIndex[x-2][y]];
181             v3 = signedDistance[mesh.xyToIndex[x][y]] -
                signedDistance[mesh.xyToIndex[x-1][y]];
182
183             // Approximate derivatives outside of domain.
184             v4 = v3;
185             v5 = v3;
186         }
```

```
187
188         // One node to left of right-hand edge.
189         else if (x == (mesh.width-1))
190         {
191             v1 = signedDistance[mesh.xyToIndex[x-2][y]] -
                    signedDistance[mesh.xyToIndex[x-3][y]];
192             v2 = signedDistance[mesh.xyToIndex[x-1][y]] -
                    signedDistance[mesh.xyToIndex[x-2][y]];
193             v3 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x-1][y]];
194             v4 = signedDistance[mesh.xyToIndex[x+1][y]] -
                    signedDistance[mesh.xyToIndex[x][y]];
195
196             // Approximate derivatives outside of domain.
197             v5 = v4;
198         }
199
200         // Node on left-hand edge.
201         else if (x == 0)
202         {
203             v5 = signedDistance[mesh.xyToIndex[2][y]] -
                    signedDistance[mesh.xyToIndex[1][y]];
204             v4 = signedDistance[mesh.xyToIndex[1][y]] -
                    signedDistance[mesh.xyToIndex[0][y]];
205
206             // Approximate derivatives outside of domain.
207             v3 = v4;
208             v2 = v4;
209             v1 = v4;
210         }
211
212         // One node to right of left-hand edge.
213         else if (x == 1)
214         {
215             v5 = signedDistance[mesh.xyToIndex[3][y]] -
                    signedDistance[mesh.xyToIndex[2][y]];
216             v4 = signedDistance[mesh.xyToIndex[2][y]] -
                    signedDistance[mesh.xyToIndex[1][y]];
217             v3 = signedDistance[mesh.xyToIndex[1][y]] -
                    signedDistance[mesh.xyToIndex[0][y]];
218
```

```
219             // Approximate derivatives outside of domain.
220             v2 = v3;
221             v1 = v3;
222         }
223
224         // Two nodes to right of left-hand edge.
225         else if (x == 2)
226         {
227             v5 = signedDistance[mesh.xyToIndex[4][y]] -
                    signedDistance[mesh.xyToIndex[3][y]];
228             v4 = signedDistance[mesh.xyToIndex[3][y]] -
                    signedDistance[mesh.xyToIndex[2][y]];
229             v3 = signedDistance[mesh.xyToIndex[2][y]] -
                    signedDistance[mesh.xyToIndex[1][y]];
230             v2 = signedDistance[mesh.xyToIndex[1][y]] -
                    signedDistance[mesh.xyToIndex[0][y]];
231
232             // Approximate derivatives outside of domain.
233             v1 = v2;
234         }
235
236         // Node lies in bulk.
237         else
238         {
239             v1 = signedDistance[mesh.xyToIndex[x-2][y]] -
                    signedDistance[mesh.xyToIndex[x-3][y]];
240             v2 = signedDistance[mesh.xyToIndex[x-1][y]] -
                    signedDistance[mesh.xyToIndex[x-2][y]];
241             v3 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x-1][y]];
242             v4 = signedDistance[mesh.xyToIndex[x+1][y]] -
                    signedDistance[mesh.xyToIndex[x][y]];
243             v5 = signedDistance[mesh.xyToIndex[x+2][y]] -
                    signedDistance[mesh.xyToIndex[x+1][y]];
244         }
245
246         double gradLeft = sign * gradHJWENO(v1, v2, v3, v4, v5);
247
248         // Upward derivatives.
249
250         // Node on bottom edge.
```

```
251         if (y == 0)
252         {
253             v1 = signedDistance[mesh.xyToIndex[x][3]] -
                    signedDistance[mesh.xyToIndex[x][2]];
254             v2 = signedDistance[mesh.xyToIndex[x][2]] -
                    signedDistance[mesh.xyToIndex[x][1]];
255             v3 = signedDistance[mesh.xyToIndex[x][1]] -
                    signedDistance[mesh.xyToIndex[x][0]];
256 
257             // Approximate derivatives outside of domain.
258             v4 = v3;
259             v5 = v3;
260         }
261 
262         // One node above bottom edge.
263         else if (y == 1)
264         {
265             v1 = signedDistance[mesh.xyToIndex[x][4]] -
                    signedDistance[mesh.xyToIndex[x][3]];
266             v2 = signedDistance[mesh.xyToIndex[x][3]] -
                    signedDistance[mesh.xyToIndex[x][2]];
267             v3 = signedDistance[mesh.xyToIndex[x][2]] -
                    signedDistance[mesh.xyToIndex[x][1]];
268             v4 = signedDistance[mesh.xyToIndex[x][1]] -
                    signedDistance[mesh.xyToIndex[x][0]];
269 
270             // Approximate derivatives outside of domain.
271             v5 = v4;
272         }
273 
274         // Node is on top edge.
275         else if (y == mesh.height)
276         {
277             v5 = signedDistance[mesh.xyToIndex[x][y-1]] -
                    signedDistance[mesh.xyToIndex[x][y-2]];
278             v4 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x][y-1]];
279 
280             // Approximate derivatives outside of domain.
281             v3 = v4;
282             v2 = v4;
```

```
283            v1 = v4;
284        }
285
286        // One node below top edge.
287        else if (y == (mesh.height - 1))
288        {
289            v5 = signedDistance[mesh.xyToIndex[x][y-1]] -
                   signedDistance[mesh.xyToIndex[x][y-2]];
290            v4 = signedDistance[mesh.xyToIndex[x][y]] -
                   signedDistance[mesh.xyToIndex[x][y-1]];
291            v3 = signedDistance[mesh.xyToIndex[x][y+1]] -
                   signedDistance[mesh.xyToIndex[x][y]];
292
293            // Approximate derivatives outside of domain.
294            v2 = v3;
295            v1 = v3;
296        }
297
298        // Two nodes below top edge.
299        else if (y == (mesh.height - 2))
300        {
301            v5 = signedDistance[mesh.xyToIndex[x][y-1]] -
                   signedDistance[mesh.xyToIndex[x][y-2]];
302            v4 = signedDistance[mesh.xyToIndex[x][y]] -
                   signedDistance[mesh.xyToIndex[x][y-1]];
303            v3 = signedDistance[mesh.xyToIndex[x][y+1]] -
                   signedDistance[mesh.xyToIndex[x][y]];
304            v2 = signedDistance[mesh.xyToIndex[x][y+2]] -
                   signedDistance[mesh.xyToIndex[x][y+1]];
305
306            // Approximate derivatives outside of domain.
307            v1 = v2;
308        }
309
310        // Node lies in bulk.
311        else
312        {
313            v1 = signedDistance[mesh.xyToIndex[x][y+3]] -
                   signedDistance[mesh.xyToIndex[x][y+2]];
314            v2 = signedDistance[mesh.xyToIndex[x][y+2]] -
                   signedDistance[mesh.xyToIndex[x][y+1]];
```

```
315             v3 = signedDistance[mesh.xyToIndex[x][y+1]] -
                    signedDistance[mesh.xyToIndex[x][y]];
316             v4 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x][y-1]];
317             v5 = signedDistance[mesh.xyToIndex[x][y-1]] -
                    signedDistance[mesh.xyToIndex[x][y-2]];
318         }
319 
320         double gradUp = sign * gradHJWENO(v1, v2, v3, v4, v5);
321 
322         // Downward derivative.
323 
324         // Node on top edge.
325         if (y == mesh.height)
326         {
327             v1 = signedDistance[mesh.xyToIndex[x][y-2]] -
                    signedDistance[mesh.xyToIndex[x][y-3]];
328             v2 = signedDistance[mesh.xyToIndex[x][y-1]] -
                    signedDistance[mesh.xyToIndex[x][y-2]];
329             v3 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x][y-1]];
330 
331             // Approximate derivatives outside of domain.
332             v4 = v3;
333             v5 = v3;
334         }
335 
336         // One node below top edge.
337         else if (y == (mesh.height - 1))
338         {
339             v1 = signedDistance[mesh.xyToIndex[x][y-2]] -
                    signedDistance[mesh.xyToIndex[x][y-3]];
340             v2 = signedDistance[mesh.xyToIndex[x][y-1]] -
                    signedDistance[mesh.xyToIndex[x][y-2]];
341             v3 = signedDistance[mesh.xyToIndex[x][y]] -
                    signedDistance[mesh.xyToIndex[x][y-1]];
342             v4 = signedDistance[mesh.xyToIndex[x][y+1]] -
                    signedDistance[mesh.xyToIndex[x][y]];
343 
344             // Approximate derivatives outside of domain.
345             v5 = v4;
```

```
346         }
347
348         // Node lies on bottom edge
349         else if (y == 0)
350         {
351             v5 = signedDistance[mesh.xyToIndex[x][2]] -
                    signedDistance[mesh.xyToIndex[x][1]];
352             v4 = signedDistance[mesh.xyToIndex[x][1]] -
                    signedDistance[mesh.xyToIndex[x][0]];
353
354             // Approximate derivatives outside of domain.
355             v3 = v4;
356             v2 = v4;
357             v1 = v4;
358         }
359
360         // One node above bottom edge.
361         else if (y == 1)
362         {
363             v5 = signedDistance[mesh.xyToIndex[x][3]] -
                    signedDistance[mesh.xyToIndex[x][2]];
364             v4 = signedDistance[mesh.xyToIndex[x][2]] -
                    signedDistance[mesh.xyToIndex[x][1]];
365             v3 = signedDistance[mesh.xyToIndex[x][1]] -
                    signedDistance[mesh.xyToIndex[x][0]];
366
367             // Approximate derivatives outside of domain.
368             v2 = v3;
369             v1 = v3;
370         }
371
372         // Two nodes above bottom edge.
373         else if (y == 2)
374         {
375             v5 = signedDistance[mesh.xyToIndex[x][4]] -
                    signedDistance[mesh.xyToIndex[x][3]];
376             v4 = signedDistance[mesh.xyToIndex[x][3]] -
                    signedDistance[mesh.xyToIndex[x][2]];
377             v3 = signedDistance[mesh.xyToIndex[x][2]] -
                    signedDistance[mesh.xyToIndex[x][1]];
378             v2 = signedDistance[mesh.xyToIndex[x][1]] -
```

```
            signedDistance[mesh.xyToIndex[x][0]];
379
380         // Approximate derivatives outside of domain.
381         v1 = v2;
382     }
383
384     // Node lies in bulk.
385     else
386     {
387         v1 = signedDistance[mesh.xyToIndex[x][y-2]] -
                signedDistance[mesh.xyToIndex[x][y-3]];
388         v2 = signedDistance[mesh.xyToIndex[x][y-1]] -
                signedDistance[mesh.xyToIndex[x][y-2]];
389         v3 = signedDistance[mesh.xyToIndex[x][y]] -
                signedDistance[mesh.xyToIndex[x][y-1]];
390         v4 = signedDistance[mesh.xyToIndex[x][y+1]] -
                signedDistance[mesh.xyToIndex[x][y]];
391         v5 = signedDistance[mesh.xyToIndex[x][y+2]] -
                signedDistance[mesh.xyToIndex[x][y+1]];
392     }
393
394     double gradDown = sign * gradHJWENO(v1, v2, v3, v4, v5);
395
396     // Compute gradient using upwind scheme.
397
398     if (gradDown > 0) grad += gradDown * gradDown;
399     if (gradLeft > 0) grad += gradLeft * gradLeft;
400     if (gradUp < 0)   grad += gradUp * gradUp;
401     if (gradRight < 0) grad += gradRight * gradRight;
402
403     grad = sqrt(grad);
404   }
405
406   // Return gradient.
407   return grad;
408 }
```

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:

```
//! Excecute Fast Marching for reinitialisation of the signed distance
    function.
/*! \param signedDistance_
        The nodal signed distance function (level set).
 */
void FastMarchingMethod::march(std::vector<double>& signedDistance_)
{
    signedDistance = &signedDistance_;
    isVelocity = false;

    // 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();
}

//! Excecute Fast Marching for velocity extension.
/*! \param signedDistance_
        The nodal signed distance function (level set).

    \param velocity_
        The nodal velocities.
 */
void FastMarchingMethod::march(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.
     */

```

```
39      signedDistance = &signedDistance_;
40      velocity = &velocity_;
41      isVelocity = true;
42
43      // Initialise the set of frozen boundary nodes.
44      initialiseFrozen();
45
46      // Initialise the heap data structure.
47      initialiseHeap();
48
49      // Initialise the set of trial nodes adjacent to the boundary.
50      initialiseTrial();
51
52      // Find the fast marching solution.
53      solve();
54
55      // Restore the original signed distance function. Only update
            velocities.
56      (*signedDistance) = signedDistanceCopy;
57  }
```