# Documentation and Tutorial for M2DO v0.01:

# Multiscale and Multiphysics Design Optimization

M2DO Lab[1,2]

[1]Cardiff University

[2]University of California, San Diego

November 2017

# Users' Guide

## Software components

The M2DO 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 M2DO 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 M2DO 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 M2DO releases. The key elements in the M2DO software suite are briefly described below for the current release, but note that modules may be added and removed with future development.

- **M2DO_FEA (Finite Element Analysis Code)**: Solves direct, adjoint, and linearized problems for the static, vibration, homogenization analysis, among many others. It uses an area fraction fixed grid finite element method.
- **M2DO_LSM**: (Level Set Method Code): Solves interface movement problem.

## Download

M2DO is available for download under the GNU Lesser General Public License (LGPL) v2.1. Please refer to the License page for terms and conditions.

**From Gitlab**

Using a git client you may clone into the repository. On a Linux/Unix/Mac system with the standard git client, this can be done by executing

git clone https://gitlab.com/stow8662/m2do-release_v01

You may also browse the code on gitlab directly. A link on the right hand side provides the option to download the code repository as a ZIP file.

https://gitlab.com/stow8662/m2do-release_v01/blob/master/m2do-release_v01.tar.gz

# Installation

M2DO has been designed with ease of installation and use in mind. This means that, wherever possible, a conscious effort was made to develop in-house code components rather than relying on third-party packages or libraries. In simple cases (serial version with no external libraries), the finite element solver can be compiled and executed with just a C++ compiler. However, the capabilities of M2DO can be extended using externally-provided software. Again, to facilitate ease of use and to promote the open source nature, whenever external software is required within the M2DO suite, packages that are free or open source have been favoured. These dependencies and third-party packages are discussed below.

**Command Line Terminal**

In general, all M2DO execution occurs via command line arguments within a terminal. For Unix/Linux or Mac OS X users, the native terminal applications are needed.

**Data Visulisation**

Users of M2DO need a data visualization tool to post-process solution files. The software currently supports .vtk output format natively read by ParaView. ParaView provides full functionality for data visualization and is freely available under an open source license. Some M2DO results are also output to .txt files, which can be read by a number of software packages, e.g. Matlab. The two most typical packages used by the development team are the following:

- ParaView
- Matlab

# Execution

Once downloaded and installed, M2DO will be ready to run simulations and design problems. Using simple command line syntax, users can execute the individual C++ programs while specifying the problem parameters in the all-purpose configuration

file. For users seeking to utilize the more advanced features of the suite (such as material microstructure design), Scripts that automate more complex tasks are available. Appropriate syntax and information for running the C++ modules and python scripts can be found below.

Run a simulation:

"make"

Output results:

"./a.out"

Clean existing compilation files:

"make clean"

# Post processing

M2DO is capable of outputting solution files and other result files that can be visualized in ParaView (.vtk).

At the end of each iteration (or at a a frequency specified by the user), M2DO will output several files that contain all of the necessary information for post-processing of results, visualization, and a restart. The restart files can then be used as input to generate the visualization files. It need to be done manually.

For a typical topology optimization analysis, these files might look like the following:

- **area.vtk** or **area.txt**: full area fraction solution.
- **level_set.vtk** or **level_set.txt**: full signed distance solution for each iteration's topology.
- **boundary_segment.txt**: file containing values for boundary segments of the geometry.
- **history.txt**: file containing the convergence history information.

# Tutorial: compliance minimization problem

## Goals

Upon completing this tutorial, the user will be familiar with performing a topology optimization for a mean compliance minimization problem. The solution will provide a cantilever beam and a simply supported beam, which can be compared to the solution from other topology optimization approaches, e.g. solid isotropic material with penalization, bi-directional evolutionary procedure, as a validation case for M2DO. Consequently, the following capabilities of M2D0 will be showcased in this tutorial:

- Finite element analysis of a structure with area fraction fixed grid finite element method;
- Shape sensitivity analysis of a structure;
- Implicit function, i.e. signed distance field, based description of a structure;
- Topology optimization with level set method.

The intent of this tutorial is to introduce a common test case which is used to explain how different equations can be implemented in M2DO. We also introduce some details on the numerics and illustrates their changes on final solution.

## Resources

The resources for this tutorial can be found in the folder **compliance_minimization** in m2do-release_v01/projects directory.

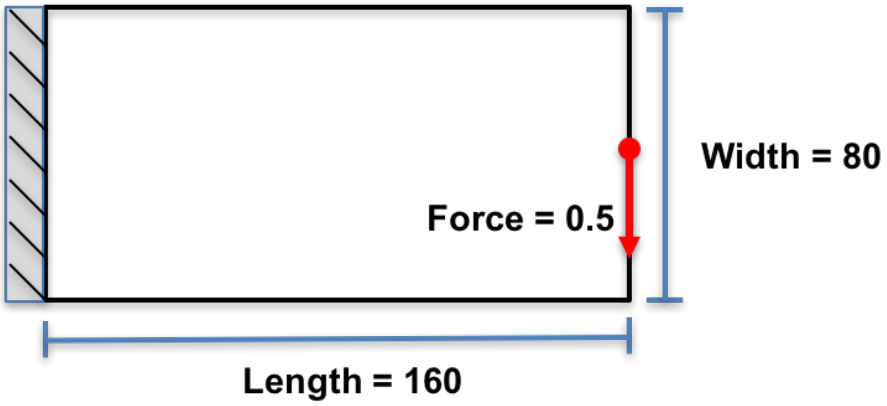


Figure 1: Configuration of the cantilever beam

# Tutorial

The following tutorial will walk you through the steps required when solving for the compliance minimization problem using M2DO. It is assumed you have already obtained the M2DO code. If you have yet to complete these requirements, please see the Download and Installation pages.

# Background

This example uses a 2D cantilever beam under a point load with configuration shown in Figure 1. It is meant to be an illustration for a structure under static load.

# Main program

The program is divided into three parts including (1) setting for finite element analysis, (2) setting for level set method, and (3) conducting iterative calculation for level set topology optimization. Details for each parts are explained below.

### *Setting for finite element analysis - lines 28 - 139*

The optimization domain is assumed to be rectangular and split into square finite elements with size of $1 \times 1$. Note that other element sizes are also allowed, but special care should be taken to establish mapping between finite element mesh and level set mesh. There are $nelx$ elements along the horizontal direction and $nely$ elements along the vertical direction as shown in Figure 2. The total number of elements is $N = nelx \times nely$. The finite element mesh model with information of mesh nodes and elements is stored in a mesh object defined in line 50. The rectangular design domain is defined in line 59, and it is meshed into square elements in line 65 as

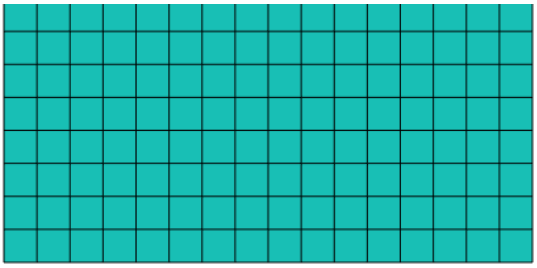

Figure 2: FEA and LSM meshing of design domain

structured mesh. The degrees of freedom for each node is assigned by calling the function in line 67. Material properties including Young's modulus, Poisson's ratio and density are input in lines 73-74. Dirichlet boundary conditions are defined in lines 88 - 89 and line 104, and Neumann boundary conditions are defined in lines 112-121 and lines 135 - 136. For this cantilever beam example, the left edge is fixed and a point load is applied at middle point on the right edge. To apply Dirichlet boundary condition, the nodes on left edge are selected first by defining a box selection area in line 88, the degrees of freedom for the selected nodes are fixed in line 89. To apply the point load, the point is selected according to its coordinate in line 112, and the corresponding degrees of freedom are selected. Magnitude of load at each direction, with one for horizontal direction and another for vertical direction, is assigned in lines 116-121. In this example, only a stationary study is required. A stationary study is thus defined in line 80. A shape sensitivity analysis is essential in this study, and it is thus declared in line 139.

### *Setting for level set method - lines 141 - 244*

Basic parameters related to level set method need to be set up first. Line 153 is for setting move limit or CFL condition. Line 157 is to set the allowed maximum iterations. Line 166 is to give area constraint. A level set mesh is created in line 172. The design domain is descritized with square elements. It is descritized into $nelx$ and $nely$ elements in the horizontal and vertical directions, respectively. Hence, a level-set array of $(nelx + 1) \times (nely + 1)$ grid points are created. Initial design with given holes' size and positions is created in lines 177-203, and it is attached to the level set mesh through line 210. For each node, the level set function is calculated as the minimum Euclidean distance. For nodes on solid element the sign is chosen to be negative, whereas for nodes on void elements the sign is chosen to be positive. The level set value each node is validated through checking its property of signed distance in line 216. The boundary of the structure is to be stored in an object named boundary in line 219.

### *Performing iterative calculation for the optimization - lines 246 - 472*

In this part of program, the iteration loop to perform the optimization is carried out, and a convergence check is also included to terminate the iteration when satisfactory solution is obtained.

Iterations are counted with $n_iterations$ and continue for a maximum of $maxit$, e.g. 300 defined in line 157. Inside the iteration loop, the boundary of the structure defined by iso-contour (2D) or surface (3D), e.g. zero level set, is discretised by finding intersection points on mesh grid with the use of marching square algorithm (line 294). The area fraction of each level set element is then calculated in line 297. The area fractions of elements are assigned to the finite elements in lines 300-313. The area fraction fixed grid finite element method is called to conduct finite element analysis through assembling global stiffness matrix (line 318) and solving finite element equation (line 325). Line 328 calculates the shape sensitivity of the compliance at each gauss points in finite element. Lines 331-346 calculate shape sensitivities for boundary points by extrapolating or interpolating from the information at gauss points with the use of the weighted least square method. Line 358 imposes the volume constraint. By completing these necessary inputs, the Lagrangian Multiplier method is applied to solve the optimization problem in lines 368 - 378.

With the obtaining of *Lambda*, the level-set function is ready to be evolved. In other words, the structural boundary can be updated to find the new structure (line 387). The up-wind finite difference scheme is used to realize this. To do so, velocities and gradients at each grid points are required. From solving the optimization equation, it only enables to compute the velocities at points along the structural boundary. In order to update the level set function, velocity values are required at all grid nodes. Line 381 thus extends or extrapolates the velocities to grid points from boundary points using the fast marching method. In practice, velocities are only extended to nodes in narrow band. Similarly, the gradients are computed in line 384. Hence, the level-set function is able to be updated (line387).

As the level-function is only updated for nodes in the narrow band for efficiency, the property of signed distance is not maintained for the rest nodes. Thus, it is important to ensure the level-set function to preserve the property of signed distance for the accuracy in solving the evolution equation. However, it may be necessary to reinitialise the level-set function too often. Currently, reinitialization at every 20 iterations is default. *nRinit* is used to count iteration. When it is reached, the reinitialization function is called in line 395 to solve the Eikonal equation.

A convergence check (lines 421 - 433) may terminate the algorithm before allowed maximum iterations are reached. The convergence check is not performed for the

first five iterations of the algorithm (line 422). After these first five iterations, the optimization terminates if the previous five objective function values are all within a tolerance of $1 \times 10^{-3}$ comparing with the current objective value and the volume is within $1 \times 10^{-4}$ of the required value $maxArea$ (lines 469 - 471).

### Outputs

The code allows to output different results, namely, area fraction, signed distance, objective function value and constraint value, from the calculation, into various formats of files. Basically, the initial and final designs, and the convergence history of objective function and constraint values are output, while the users are given the option to output each step's results during iterative calculations. Lines 257-271 write initial design's area fractions and signed distances into vtk and txt files, and their counterparts for final solution are written into files in lines 454 - 467. If the users prefer to write each iteration's results, they only need to active the flag in line 274.

## Running optimization

To run the code, the following command lines should be implemented in terminal for compilation

make

and outputting results

./bin/a.out

They can be implemented in a single command line:

make && ./bin/a.out

## Results

For a given initial design as shown in Figure 3a, the structure converges to an optimal solution as given in Figure 3b after 82 iterations with a compliance value of 15.1 by using default parameters and convergence criterion given in the source file. The convergence history is illustrated in Figure 4.

(a) Initial design (b) Initial design

Figure 3: Initial design and optimal solution for the cantilever example

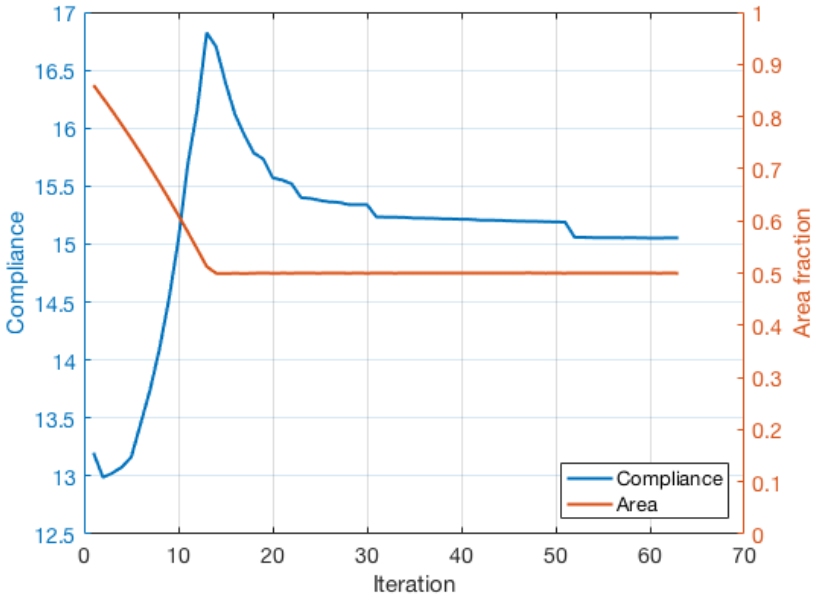


Figure 4: Convergence history of compliance value and area fraction for the cantilever example

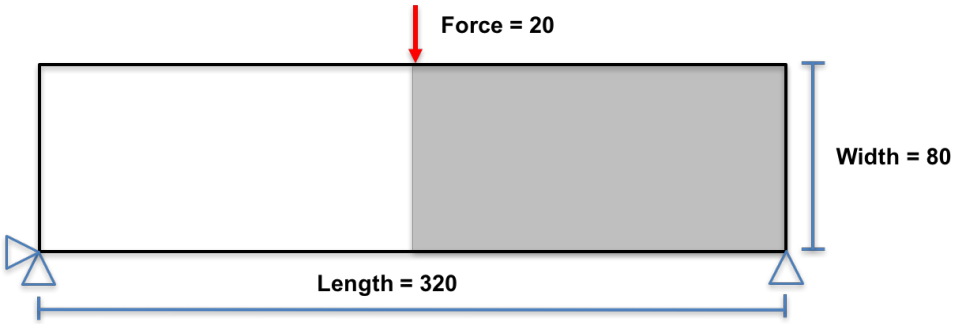


Figure 5: Configuration of the MBB beam

(a) Initial design (b) Initial design

Figure 6: Initial design and optimal solution for the MBB example

## Illustration of the extension to other boundary condition with a MBB beam example

The code can be easily changed to consider different boundary conditions, loads, and different initial design. A simple extension to find optimal design for a simply supported beam or MBB beam as shown in Figure 5 is demonstrated here. Only half of the beam needs to be solved due to symmetry of load and boundary conditions about the vertical axis. The right half (the shaded area in the figure) is considered in this example. The configuration of the half beam is set the same as the previous cantilever beam. Hence, it only needs to change the boundary conditions and loads. The horizontal translation of the left edge of the half beam need to be restricted, and the vertical motion of lower right-hand conner is not allowed. These boundary conditions are realized in lines 93-102. To apply the vertical point load for the node at the upper left-hand conner, corresponding node and related degree of freedom need to be selected and the magnitude of the load is assigned in lines 125 - 133. With these setting, the optimization for MBB can be solved. For the given initial design as shown in Figure 6a, the structure converges to an optimal solution as given in Figure 6b after 165 iterations with a compliance value of 7492.2 by using default parameters and convergence criterion given in the source file. The convergence history is illustrated in Figure 7.

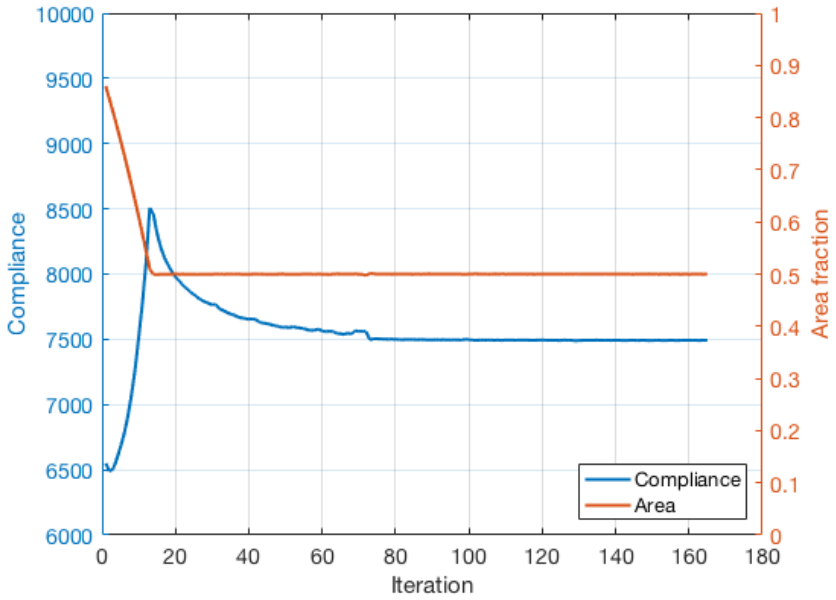


Figure 7: Convergence history of compliance value and area fraction for the MBB example

# C++ code for compliance minimization of a cantilever beam

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

#include "MatrixM2DO.h"
//#include "VectorM2DO.h"

using namespace std ;

namespace FEA = M2DO_FEA ;
namespace LSM = M2DO_LSM ;

#include <time.h>
#include <sys/time.h>
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        //  Handle error
        return 0;
    }
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}
double get_cpu_time(){
    return (double)clock() / CLOCKS_PER_SEC;
}

int main () {

    ////////////////////////////////////////////////////////////////////////
    //                      //
```

```
30      //    Part 1: SETTING FOR FINITE ELEMENT ANALYSIS            //
31      //                        //
32      ////////////////////////////////////////////////////////////////////////
33
34      /*
35       Dimensionality of problem:
36       */
37
38      const int spacedim = 2 ;
39
40      /*
41       FEA & level set mesh parameters:
42       */
43
44      const unsigned int nelx = 160, nely = 80;
45
46      /*
47       Create an FEA mesh object.
48       */
49
50      FEA::Mesh fea_mesh (spacedim) ;
51
52      /*
53       Mesh a hyper rectangle.
54       */
55
56      Matrix<double,-1,-1> fea_box (4, 2) ;
57
58      // define design domain
59      fea_box.data = {{0,0},{nelx,0},{nelx,nely},{0,nely}};
60
61      // mesh size in horizontal and vertical directions
62      vector<int> nel = {nelx, nely} ;
63
64      int element_order = 2 ;
65      fea_mesh.MeshSolidHyperRectangle (nel, fea_box, element_order, false) ;
66      fea_mesh.is_structured = true ;
67      fea_mesh.AssignDof () ;
68
69      /*
70       Add material properties:
71       */
72
73      double E = 1.0, v = 0.3, rho = 1.0;
74      fea_mesh.solid_materials.push_back (FEA::SolidMaterial (spacedim, E, v, rho)) ;
75
76      /*
77       Next we specify that we will undertake a stationary study, which takes the
             form [K]{u} = {f}.
78       */
79
80      FEA::StationaryStudy fea_study (fea_mesh) ;
81
82      /*
83       Add a homogeneous Dirichlet boundary condition (fix some nodes).
84       */
85
86      // Example 1: cantilever beam
87
```

```
88      // vector<int> fixed_nodes = fea_mesh.GetNodesByCoordinates ({0.0, 0.0}, {0.1,
            1.0E10}) ;
89      // vector<int> fixed_dof = fea_mesh.dof (fixed_nodes) ;
90
91      // Example 2: half of simply supported beam or Messerschmitt-Bolkow-Blohm (MBB)
             beam
92
93      vector<int> fixed_nodes_left = fea_mesh.GetNodesByCoordinates ({0.0, 0.0},
            {0.1, 1.0E10}) ;
94      vector<int> fixed_dof_left = fea_mesh.dof (fixed_nodes_left, {0}) ;
95
96      vector<int> fixed_nodes_right = fea_mesh.GetNodesByCoordinates ({nelx, 0.0},
            {0.1, 0.1}) ;
97      vector<int> fixed_dof_right = fea_mesh.dof (fixed_nodes_right, {1}) ;
98
99      vector<int> fixed_dof;
100     fixed_dof.reserve( fixed_dof_left.size() + fixed_dof_right.size() );
101     fixed_dof.insert( fixed_dof.end(), fixed_dof_left.begin(), fixed_dof_left.end()
             );
102     fixed_dof.insert( fixed_dof.end(), fixed_dof_right.begin(), fixed_dof_right.end
            () );
103
104     fea_study.AddBoundaryConditions (FEA::HomogeneousDirichletBoundaryConditions (
            fixed_dof, fea_mesh.n_dof)) ;
105
106     /*
107      Apply a point load.
108      */
109
110     // Example 1: cantilever beam
111
112     // vector<int> load_node = fea_mesh.GetNodesByCoordinates ({1.0*nelx, 0.5*nely
            }, {1e-12, 1e-12}) ;
113     // vector<int> load_dof  = fea_mesh.dof (load_node) ;
114     // vector<double> load_val (load_node.size() * 2) ;
115
116     // for (int i = 0 ; i < load_node.size() ; ++i) {
117
118     //    load_val[2*i]   = 0.00 ;
119     //    load_val[2*i+1] = -0.5 ;
120
121     // }
122
123     // Example 2: half of simply supported beam or Messerschmitt-Bolkow-Blohm (MBB)
             beam
124
125     vector<int> load_node = fea_mesh.GetNodesByCoordinates ({0, nely}, {1e-12, 1e
            -12}) ;
126     vector<int> load_dof  = fea_mesh.dof (load_node, {1}) ;
127     vector<double> load_val (load_node.size()) ;
128
129     for (int i = 0 ; i < load_node.size() ; ++i) {
130
131         load_val[i] = -10.0 ;
132
133     }
134
135     FEA::PointValues point_load (load_dof, load_val) ;
136     fea_study.AssembleF (point_load, false) ;
```

```
137
138     // Create sensitivity analysis instance.
139     FEA::SensitivityAnalysis sens(fea_study) ;
140
141     ////////////////////////////////////////////////////////////////////////
142     //                      //
143     //     Part 2: SETTING FOR LEVEL SET METHOD         //
144     //                      //
145     ////////////////////////////////////////////////////////////////////////
146
147     /*
148      Tread carefully; these be Renato's additions (very funny!).
149      */
150
151     // Maximum displacement per iteration, in units of the mesh spacing.
152     // This is the CFL limit.
153     double moveLimit = 0.5 ;
154
155     // Set maximum running time.
156     double maxTime = 6000 ;
157     int maxit = 300;
158
159     // Set sampling interval.
160     double sampleInterval = 50 ;
161
162     // Set time of the next sample.
163     double nextSample = 50 ;
164
165     // Maximum material area.
166     double maxArea = 0.5 ;
167
168     // Default temperature of the thermal bath.
169     double temperature = 0 ;
170
171     // Initialise the level set mesh (same resolution as the FE mesh).
172     LSM::Mesh lsmMesh(nelx, nely, false) ;
173
174     double meshArea = lsmMesh.width * lsmMesh.height ;
175
176     // Create two horizontal rows with four equally space holes.
177     vector<LSM::Hole> holes ;
178
179     holes.push_back(LSM::Hole(16, 14, 5)) ;
180     holes.push_back(LSM::Hole(32, 27, 5)) ;
181     holes.push_back(LSM::Hole(48, 14, 5)) ;
182     holes.push_back(LSM::Hole(64, 27, 5)) ;
183     holes.push_back(LSM::Hole(80, 14, 5)) ;
184     holes.push_back(LSM::Hole(96, 27, 5)) ;
185     holes.push_back(LSM::Hole(112, 14, 5)) ;
186     holes.push_back(LSM::Hole(128, 27, 5)) ;
187     holes.push_back(LSM::Hole(144, 14, 5)) ;
188
189     holes.push_back(LSM::Hole(16, 40, 5)) ;
190     holes.push_back(LSM::Hole(32, 53, 5)) ;
191     holes.push_back(LSM::Hole(48, 40, 5)) ;
192     holes.push_back(LSM::Hole(64, 53, 5)) ;
193     holes.push_back(LSM::Hole(80, 40, 5)) ;
194     holes.push_back(LSM::Hole(96, 53, 5)) ;
195     holes.push_back(LSM::Hole(112, 40, 5)) ;
```

```
196     holes.push_back(LSM::Hole(128, 53, 5)) ;
197     holes.push_back(LSM::Hole(144, 40, 5)) ;
198
199     holes.push_back(LSM::Hole(16, 66, 5)) ;
200     holes.push_back(LSM::Hole(48, 66, 5)) ;
201     holes.push_back(LSM::Hole(80, 66, 5)) ;
202     holes.push_back(LSM::Hole(112, 66, 5)) ;
203     holes.push_back(LSM::Hole(144, 66, 5)) ;
204
205     // Initialise guess solution for cg
206     int n_dof = fea_mesh.n_dof ;
207     std::vector<double> u_guess(n_dof,0.0);
208
209     // Initialise the level set object (from the hole vector).
210     LSM::LevelSet levelSet(lsmMesh, holes, moveLimit, 6, false) ;
211
212     // Initialise io object.
213     LSM::InputOutput io ;
214
215     // Reinitialise the level set to a signed distance function.
216     levelSet.reinitialise() ;
217
218     // Initialise the boundary object.
219     LSM::Boundary boundary(levelSet) ;
220
221     // Initialise random number generator.
222     LSM::MersenneTwister rng ;
223
224     // Number of cycles since signed distance reinitialisation.
225     unsigned int nReinit = 0 ;
226
227     // Running time.
228     double time = 0 ;
229
230     // Time measurements.
231     vector<double> times ;
232
233     // Compliance measurements.
234     vector<double> compliances ;
235
236     // Boundary curvature measurements.
237     vector<double> areas ;
238
239     /* Lambda values for the optimiser.
240      These are reused, i.e. the solution from the current iteration is
241      used as an estimate for the next, hence we declare the vector
242      outside of the main loop.
243      */
244     vector<double> lambdas(2) ;
245
246     ////////////////////////////////////////////////////////////////////
247     //                      //
248     //   Part 3: ITERATION FOR LEVEL SET TOPOLOGY OPTIMIZATION      //
249     //                      //
250     ////////////////////////////////////////////////////////////////////
251
252     // setup for outputing convergence history of objective function values and
            constraints
253     ofstream convergenceHistory;
```

```
254     convergenceHistory.open("convergenceHistory.txt",ofstream::out);
255
256     // write initial design into vtk & txt format - signed distance
257     std::ostringstream fileNameLSF, fileNameArea;
258     fileNameLSF.str("");
259     fileNameLSF << "level-set-initial.vtk";
260     io.saveLevelSetVTK(fileNameLSF, levelSet);
261     fileNameLSF.str("");
262     fileNameLSF << "level-set-initial.txt";
263     io.saveLevelSetTXT(fileNameLSF, levelSet, true);
264
265     // write optimal design into vtk & txt format - area fraction
266     fileNameArea.str("");
267     fileNameArea << "area-initial.vtk";
268     io.saveAreaFractionsVTK(fileNameArea, lsmMesh);
269     fileNameArea.str("");
270     fileNameArea << "area-initial.txt";
271     io.saveAreaFractionsTXT(fileNameArea, lsmMesh);
272
273     // flag for chosing whether write level set function and area fraction into vtk
            files or not
274     bool flagOutputHistory = false;
275
276     // Integrate until we exceed the maximum time.
277     int n_iterations = 0 ;
278
279     // Initialise vector to save objective history.
280     std::vector<double> Objective_Values;
281     // Initialise variable to stop loop.
282     double Relative_Difference = 1.0;
283
284     // iterative calculation
285
286     cout << "\nStarting compliance minimisation demo...\n\n" ;
287
288     while (n_iterations < maxit) {
289
290         ++n_iterations ;
291         //cout << "Echo 1" << endl;
292
293         // Perform boundary discretisation.
294         boundary.discretise(false, lambdas.size()) ;
295
296         // Compute element area fractions.
297         boundary.computeAreaFractions() ;
298
299         // Assign area fractions.
300         for (unsigned int i=0 ; i< fea_mesh.solid_elements.size() ; i++) {
301
302             if (lsmMesh.elements[i].area < 1e-3) {
303
304                 fea_mesh.solid_elements[i].area_fraction = 1e-3 ;
305
306             }
307             else {
308
309                 fea_mesh.solid_elements[i].area_fraction = lsmMesh.elements[i].area
                        ;
310
```

```
311             }
312
313         }
314
315         /*
316          Assemble stiffness matrix [K] using area fraction method:
317          */
318         fea_study.Assemble_K_With_Area_Fractions_Sparse (false) ;
319
320         /*
321          Solve equation:
322          */
323
324         double cg_tolerence = 1.0e-6;
325         fea_study.Solve_With_CG (false, cg_tolerence,u_guess) ;
326
327         // Compute compliance sensitivities (stress*strain) at the Gauss points.
328         sens.ComputeComplianceSensitivities(false) ;
329
330         double abs_bsens_max = 0.0;
331         for (int i=0 ; i<boundary.points.size() ; i++) {
332
333             vector<double> boundary_point (2, 0.0) ;
334             boundary_point[0] = boundary.points[i].coord.x;
335             boundary_point[1] = boundary.points[i].coord.y;
336
337             // Interpolate Guass point sensitivities by least squares.
338             sens.ComputeBoundarySensitivities(boundary_point) ;
339
340
341             // Assign sensitivities.
342             boundary.points[i].sensitivities[0] = -sens.boundary_sensitivities[i] ;
343             boundary.points[i].sensitivities[1] = -1 ;
344
345             abs_bsens_max = std::max(abs_bsens_max, std::abs(sens.
                boundary_sensitivities[i]));
346         }
347
348         // clearing sens.boundarysens vector.
349         sens.boundary_sensitivities.clear() ;
350
351         // Time step associated with the iteration.
352         double timeStep ;
353
354         // Constraint distance vector.
355         vector<double> constraintDistances ;
356
357         // Push current distance from constraint violation into vector.
358         constraintDistances.push_back(meshArea*maxArea - boundary.area) ;
359
360         /* Initialise the optimisation object.
361
362          The Optimise class is a lightweight object so there is no cost for
363          reinitialising at every iteration. A smart compiler will optimise
364          this anyway, i.e. the same memory space will be reused. It is better
365          to place objects in the correct scope in order to aid readability
366          and to avoid unintended name clashes, etc.
367          */
368         LSM::Optimise optimise(boundary.points,  timeStep, moveLimit) ;
```

```
369
370         // set up required parameters
371         optimise.length_x = lsmMesh.width;
372         optimise.length_y = lsmMesh.height;
373         optimise.boundary_area = boundary.area; // area of structure
374         optimise.mesh_area = meshArea; // area of the entire mesh
375         optimise.max_area = maxArea; // maximum area
376
377         // Perform the optimisation.
378         optimise.Solve_With_NewtonRaphson() ;
379
380         // Extend boundary point velocities to all narrow band nodes.
381         levelSet.computeVelocities(boundary.points, timeStep, temperature, rng) ;
382
383         // Compute gradient of the signed distance function within the narrow band.
384         levelSet.computeGradients() ;
385
386         // Update the level set function.
387         bool isReinitialised = levelSet.update(timeStep) ;
388
389         // Reinitialise the signed distance function, if necessary.
390         if (!isReinitialised) {
391
392             // Reinitialise at least every 20 iterations.
393             if (nReinit == 20) {
394
395                 levelSet.reinitialise() ;
396                 nReinit = 0 ;
397             }
398
399         }
400         else nReinit = 0 ;
401
402         // Increment the number of steps since reinitialisation.
403         nReinit++ ;
404
405         // Increment the time.
406         time += timeStep;
407
408         // Calculate current area fraction.
409         double area = boundary.area / meshArea ;
410
411         // Record the time, compliance, and area.
412         times.push_back(time) ;
413         //compliances.push_back(study.compliance);
414         areas.push_back(area) ;
415         Objective_Values.push_back(sens.objective);
416
417         // Write objective function values and area into txt file
418         convergenceHistory << n_iterations << "\t" << setprecision(15) <<
                Objective_Values[n_iterations-1] << "\t" << area << endl;
419
420         // Converence criterion [Dunning_11_FINEL]
421         double Objective_Value_k, Objective_Value_m;
422         if (n_iterations > 5) {
423
424             Objective_Value_k = sens.objective;
425             Relative_Difference = 0.0;
426             for (int i = 1; i <= 5; i++) {
```

```
427
428                 Objective_Value_m = Objective_Values[n_iterations - i - 1];
429                 Relative_Difference = max(Relative_Difference, abs((
                        Objective_Value_k - Objective_Value_m)/Objective_Value_k));
430
431             }
432
433         }
434
435         if (n_iterations==1) {
436
437             // Print output header.
438             printf("--------------------------------\n");
439             printf("%8s %12s %10s\n", "Iteration", "Compliance", "Area");
440             printf("--------------------------------\n");
441
442         }
443         // Print statistics.
444         printf("%8.1f %12.4f %10.4f\n", double (n_iterations), sens.objective, area
                ) ;
445
446         // Write level set and boundary segments to file.
447         if (flagOutputHistory) {
448
449             io.saveLevelSetVTK(n_iterations, levelSet) ;
450             io.saveAreaFractionsVTK(n_iterations, lsmMesh) ;
451         }
452
453         // Write optimal design into vtk & txt format - area fraction
454         fileNameArea.str("");
455         fileNameArea << "area-optimal.vtk";
456         io.saveAreaFractionsVTK(fileNameArea, lsmMesh);
457         fileNameArea.str("");
458         fileNameArea << "area-optimal.txt";
459         io.saveAreaFractionsTXT(fileNameArea, lsmMesh);
460
461         // Write optimal design into vtk & txt format - signed distance
462         fileNameLSF.str("");
463         fileNameLSF << "level-set-optimal.vtk";
464         io.saveLevelSetVTK(fileNameLSF, levelSet);
465         fileNameLSF.str("");
466         fileNameLSF << "level-set-optimal.txt";
467         io.saveLevelSetTXT(fileNameLSF, levelSet, true);
468
469         if ((Relative_Difference < 0.0001) & (area < 1.001*maxArea)) {
470             break;
471         }
472     }
473
474     /*
475      Aaaaaand that's all, folks!
476      */
477
478     cout << "\nProgram complete.\n\n" ;
479
480     return 0 ;
481 }
```

# Theoretical Background of M2DO code

## Finite Element Analysis

### Area Fraction Weighted Fixed Grid Approach

A fixed grid is generated by superimposing a rectangular grid of equal sized elements on the given structure instead of generating a mesh to fit the structure. Some of these elements are inside of the structure (I), some are outside (O) and some are on the the boundary, namely neither-in-nor-out (NIO) elements. As O element is given a material property significantly less than an I element and the problem becomes a bimaterial one.

A NIO element is partially inside the structure and its material property value is not constant nor continuous over the element. Such an element is approximated by transforming the bimaterial element into a homogeneous isotropic element. The material property matrix of a NIO element is computed using:

$$[D(NIO)^e] = \alpha\,[D(I)^e] \tag{1}$$

where $[D(NIO)^e]$ is the elemental material property of a NIO element, $[D(I)^e]$ is for the elemental material property of inside, and $\alpha$ is the area ratio calculated by

$$\alpha = \frac{A_I}{A^e} \tag{2}$$

with $A_I$ is the area inside the structure within the NIO element, and $A^e$ is the total area of the $e$-th element.

Using these values of $[D(NIO)^e]$, the stiffness matrix can be computed and a standard finite element analysis can be applied to determine the displacements and hence stress values of elements.

## Sensitivity Analysis

The principle of virtual work, also known as the principle of virtual displacement, is stated that for any quasi-static and admissible virtual displacement from an equilibrium configuration, the increment of strain energy stored is equal to the increment of work done by body force $\{\boldsymbol{b}\}$ in volume $V$ and surface traction $\{\boldsymbol{t}\}$ on surface $S$. It is formulated as

$$\int \{\delta\boldsymbol{\varepsilon}\}^T \{\boldsymbol{\sigma}\}\, dV = \int \{\delta\boldsymbol{u}\}^T \{\boldsymbol{b}\}\, dV + \int \{\delta\boldsymbol{u}\}^T \{\boldsymbol{t}\}\, dS \tag{3}$$

where $\{\delta\boldsymbol{\varepsilon}\}$ is the vector of strains, $\{\delta\boldsymbol{u}\}$ is the virtual displacement. In textbook on topology optimization, the right and left hand sides are written as

$$\begin{aligned} a(\boldsymbol{u},\boldsymbol{v}) &= \int \{\delta\boldsymbol{\varepsilon}\}^T \{\boldsymbol{\sigma}\}\, dV \\ &= \int \{\delta\boldsymbol{\varepsilon}\}^T [\mathbf{E}] \{\boldsymbol{\varepsilon}\}\, d\Omega = \int \{\boldsymbol{\varepsilon}(\boldsymbol{v})\}^T [\mathbf{E}] \{\boldsymbol{\varepsilon}(\boldsymbol{u})\}\, d\Omega \qquad (4) \\ l(\boldsymbol{v}) &= \int \{\delta\boldsymbol{u}\}^T \{\boldsymbol{b}\}\, dV + \int \{\delta\boldsymbol{u}\}^T \{\boldsymbol{t}\}\, dS \\ &= \int \{\boldsymbol{v}\}^T \{\boldsymbol{b}\}\, dV + \int \{\boldsymbol{v}\}^T \{\boldsymbol{t}\}\, dS \qquad (5) \end{aligned}$$

where $[\mathbf{E}]$ is the constitutive matrix, $\boldsymbol{v}$ is the virtual displacement that is equivalent to $\{\delta\boldsymbol{u}\}$.

The Lagrange multiplier method is applied to solve the optimization problem, where the Lagrange function is:

$$\mathcal{L}(\Omega, \{\boldsymbol{u}\}, \{\boldsymbol{\lambda}\}) = \mathcal{J}(\Omega, \{\boldsymbol{u}\}) + a(\{\boldsymbol{u}\}, \{\boldsymbol{\lambda}\}) - l(\{\boldsymbol{\lambda}\}) \tag{6}$$

in which $\{\boldsymbol{\lambda}\}$ is the adjoint variable.

The shape derivation of the objective function is obtained by differentiating

$$\mathcal{J}(\Omega) = \mathcal{L}(\Omega, \{\boldsymbol{u}\}, \{\boldsymbol{\lambda}\}) \tag{7}$$

which, by the chain rule theorem, reduces to the partial derivative of $\mathcal{L}$ with respect to $\Omega$ in the direction $\theta$

$$\mathcal{J}'(\Omega)(\theta) = \frac{\partial \mathcal{L}}{\partial \Omega}. \tag{8}$$

Applying **Lemma** 4 and 5 in [AJT04] to $\mathcal{L}$ and substituting $\mathcal{J}$ for compliance, we

obtain:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \Omega} &= \frac{\partial}{\partial \Omega}\left(\int \{\boldsymbol{u}\}^T\{\boldsymbol{b}\}dV + \int \{\boldsymbol{u}\}^T\{\boldsymbol{t}\}dS \right. \\
&\quad + \int \{\boldsymbol{\varepsilon}(\boldsymbol{\lambda})\}^T [\mathbf{E}] \{\boldsymbol{\varepsilon}(\boldsymbol{u})\} d\Omega \\
&\quad \left. - \int \{\boldsymbol{\lambda}\}^T \{\boldsymbol{b}\} dV - \int \{\boldsymbol{\lambda}\}^T \{\boldsymbol{t}\} dS \right) \\
&= \int_{\partial\Omega} \theta \cdot n \left(\{\boldsymbol{u}\}^T\{\boldsymbol{b}\} + \{\boldsymbol{\varepsilon}(\boldsymbol{\lambda})\}^T [\mathbf{E}] \{\boldsymbol{\varepsilon}(\boldsymbol{u})\} - \{\boldsymbol{\lambda}\}^T \{\boldsymbol{b}\}\right) dS \\
&\quad + \int_{\partial\Omega} \theta \cdot n \left(\frac{\partial \{\boldsymbol{u}\}^T\{\boldsymbol{t}\}}{\partial n} + H\{\boldsymbol{u}\}^T\{\boldsymbol{t}\}\right) dS \\
&\quad - \int_{\partial\Omega} \theta \cdot n \left(\frac{\partial \{\boldsymbol{\lambda}\}^T\{\boldsymbol{t}\}}{\partial n} + H\{\boldsymbol{\lambda}\}^T\{\boldsymbol{t}\}\right) dS \qquad (9)
\end{aligned}$$

In the case of a self-adjoint problem, e.g. compliance, it has $\{\boldsymbol{\lambda}\} = -\{\boldsymbol{u}\}$.

# Level Set Method

## Implicit Surfaces

In three spatial dimensions, the lower-dimensional interface is a surface that separates $\mathcal{R}^3$ into separate subdomains with nonzero volumes. We consider only closed surfaces with clearly defined interior and exterior regions.

For complicated surfaces with no analytical representation, we again need to use a discretization. In three spatial dimensions the explicit representation can be quite difficult to discrete. One needs to choose a number of points on the two-dimensional surface and record their connectivity. If the exact surface and its connectivity are known, it is simple to tile the surface with triangles whose vertices lie on the interface and whose edges indicate connectivity. On the other hand, if connectivity is not known, it can be quite difficult to determine.

Connectivity can change for dynamic implicit surfaces, i.e., surface that are moving around.

## Signed Distance Function

A distance function $d(\boldsymbol{x})$ is defined as

$$d(\boldsymbol{x}) = \min\left(|\boldsymbol{x} - \boldsymbol{x}_I|\right), \quad \forall \boldsymbol{x}_I \in \partial\Omega, \qquad (10)$$

implying that $d(\boldsymbol{x}) = 0$ on the boundary where $\boldsymbol{x} \in \partial\Omega$. Geometrically, $d$ may be constructed as follows. If $\boldsymbol{x} \in \partial\Omega$, then $d(\boldsymbol{x}) = 0$. Otherwise, for a given point $\boldsymbol{x}$,

find the point on the boundary set $\partial\Omega$ closest to $\boldsymbol{x}$, and label this point $\boldsymbol{x}_C$. Then $d(\boldsymbol{x}) = |\boldsymbol{x} - \boldsymbol{x}_c|$.

A signed distance functions is an implicit function $\phi$ with $|\phi(\boldsymbol{x})| = d(\boldsymbol{x})$ for all $\boldsymbol{x}$. Thus, $\phi(\boldsymbol{x}) = d(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \partial\Omega$, $\phi(\boldsymbol{x}) = -d(\boldsymbol{x})$ for all $\boldsymbol{x} \in \Omega^-$, and $\phi(\boldsymbol{x}) = d(\boldsymbol{x})$ for all $\boldsymbol{x} \in \Omega^+$. In a compact for, it is written as

$$\phi(\boldsymbol{x}) = \begin{cases} -d(\boldsymbol{x}), & \forall \boldsymbol{x} \in \Omega^- \\ 0, & \forall \boldsymbol{x} \in \partial\Omega \\ d(\boldsymbol{x}), & \forall \boldsymbol{x} \in \Omega^+ \end{cases} \tag{11}$$

. Signed distance functions share all the properties of implicit functions. In addition, there are a number of new properties that only signed distance functions possess. For example,

$$|\nabla\phi| = 1, \tag{12}$$

which is known as the eikonal equation.

## Hamilton Jacobi Equation

An implicitly defined boundary of the structure used during level-set method is updated by solving pseudo time-dependent Hamilton-Jacobi equation as shown:

$$\frac{\partial\phi(x,t)}{\partial t} + \nabla\phi(x,t)\frac{dx}{dt} \tag{13}$$

where $t$ is time, $\nabla\phi(x,t)$ is the gradient of the level set function. To put the equation simply, a smooth boundary $\Gamma = x|\phi(x) = 0$ at given time $x$ and space $t$ is changed by its normal velocity.

Being a hyperbolic partial differential equation, Hamilton-Jacobi equation is often solved numerically. In practice, an upwind scheme with high-order differentiation is used with a constraint from CFL (Courant–Friedrichs–Lewy) condition is a common practice to obtain a stable solution.

## Boundary Evolution

Since the structural boundary is the zero value level set, so the boundary evolution is implicitly achieved by the updating of level set function.

In numerical implementations, the structural boundary is discretized and approximately represented by a series of boundary points which satisfies $\phi_(\boldsymbol{x}) = 0$. If two adjacent nodes have $\phi$-values with opposite signs then there is a boundary point between them, which is taken to lie on the edge between the nodes, with a position

determined by linear interpolation. The boundary points form a set of closed curves, which provide a discrete representation of the boundary.

### Update and reinitialization

The level set function in each node can be updated by the following discretized H-J equation with the use of the up-wind differential scheme:

$$\phi_i(t + \Delta t) = \phi_i(t) - v_i^n \Delta t \cdot |\nabla \phi(t)|_i \tag{14}$$

where $v_i^n \Delta t$ is the boundary movement obtained by solving a sublevel linearised programme; the gradient filed is estimated for each node using the Hamilton–Jacobi weighted essentially non-oscillatory method (HJ-WENO) described in [OF03].

Instead of updating the level set values in whole field, we only restrict the update to nodes within a narrow band close to the boundary. This improves the efficiency of the method but it means that $\phi_i$ is given by the signed distance to the boundary only within the narrow band. To correct for this effect, it is common that all of the $\phi_i$ variables are periodically reinitialised to be consistent with a signed distance function, i.e., satisfying the following equation:

$$|\nabla \phi| = 1 \tag{15}$$

This reinitialisation uses the same fast-marching implementation used for the velocity extension.

## Level Set Method Based Topology Optimization

### Sequential linear programming level set topology optimization

The velocities required for the level set update are obtained by solving an optimization problem. A generic optimization problem can be formulated using the position of the structural boundary as the design variable:

$$\begin{aligned} \underset{\Omega}{\text{minimize}} \quad & f(\Omega) \\ \text{subject to} \quad & g_i(\Omega) \leq 0 \end{aligned} \tag{16}$$

where $f(\Omega)$ is the objective function and $g_i$ is the $i^{\text{th}}$ inequality constraint function. The objective and constraint functions are linearized about the design variables at

each $k^{\text{th}}$ iteration using a first-order Taylor expansion:

$$\begin{aligned} \underset{\Delta\Omega^k}{\text{minimize}} \quad & \frac{\partial f}{\partial \Omega^k} \cdot \Delta\Omega^k \\ \text{subject to} \quad & \frac{\partial g_i}{\partial \Omega^k} \cdot \Delta\Omega^k \leq -\bar{g}_i^k \end{aligned} \tag{17}$$

where $\Delta\Omega^k$ is the update for the design domain $\Omega$ and $\bar{g}_i^k$ is the change in the $i^{\text{th}}$ constraint at iteration $k$.

In the level-set description of the boundary, shape derivatives provide information about how a function changes over time with respect to a movement of the boundary point. They usually take the form of boundary integrals [AJT04]. In this case,

$$\frac{\partial f}{\partial \Omega} \cdot \Delta\Omega = \Delta t \int_\Gamma s_f V_n d\Gamma, \tag{18}$$

$$\frac{\partial g_i}{\partial \Omega} \cdot \Delta\Omega = \Delta t \int_\Gamma s_{gi} V_n d\Gamma, \tag{19}$$

where $s_f$ and $s_{g_i}$ are the shape sensitivity functions for the objective and the $i^{\text{th}}$ constraints. Discretizing the boundary at $nb$ points, one can rewrite:

$$\frac{\partial f}{\partial \Omega} \cdot \Delta\Omega \approx \sum_{j=1}^{nb} \Delta t V_{nj} s_{f,j} l_j = \mathbf{C}_f \cdot \mathbf{V}_n \Delta t, \tag{20}$$

$$\frac{\partial g_i}{\partial \Omega} \cdot \Delta\Omega \approx \sum_{j=1}^{nb} \Delta t V_{nj} s_{gi,j} l_j = \mathbf{C}_{gi} \cdot \mathbf{V}_n \Delta t, \tag{21}$$

where $l_j$ is the discrete length of the boundary around the boundary point $j$, $\mathbf{C}_f$ and $\mathbf{C}_{gi}$ are vectors containing integral coefficients and $\mathbf{V}_n$ is the vector of normal velocities. For a constrained problem, one can write

$$\mathbf{V}_n \Delta t = \alpha \mathbf{d}, \tag{22}$$

where $\mathbf{d}$ is the search direction for the boundary update and $\alpha > 0$ is the actual distance of the boundary movement. Then, the optimization formulation to obtain the optimal boundary velocities can be written as:

$$\begin{aligned} \underset{\alpha^k, \boldsymbol{\lambda}^k}{\text{minimize}} \quad & \Delta t \mathbf{C}_f^k \cdot \mathbf{V}_n^k(\alpha^k, \boldsymbol{\lambda}^k) \\ \text{subject to} \quad & \Delta t \mathbf{C}_i^k \cdot \mathbf{V}_n^k(\alpha^k, \boldsymbol{\lambda}^k) \leq -\bar{g}_i^k \\ & \mathbf{V}_{n,\min}^k \leq \mathbf{V}_n^k \leq \mathbf{V}_{n,\max}^k \end{aligned} \tag{23}$$

where $\boldsymbol{\lambda}$ are Lagrange multipliers for each constraint function. This optimization problem is solved at every iteration $k$. More details can be found in [DK15, SDK16].

## Velocity extension

Velocity function defined in previous equation is only computed at points along the structural boundary. In order to update the level set function, velocity values at all grid points are required. Thus, the velocity function must be extended or extrapolated to grid points away from the boundary. Natural velocity extension schemes compute strain and sensitivity fields over the entire design domain. Methods that achieve this include filling the void part with a fictitious weak material [AJT04], or smoothing the velocity field over the discontinuity at the boundary edge [WW06]. However, the implicit function often becomes too steep or flat around the boundary, which leads to potential stability issues. Thus, these schemes usually require frequent reinitializing of the implicit function to a signed distance function to maintain stability [AJT04, WW06].

To avoid frequent reinitialization, we employ an extension velocity technique designed to maintain the signed distance function [AS99]. This technique ensures the preservation of the signed distance by using the efficient fast marching method to solve the following equation:

$$\nabla\phi_t \nabla V_{\text{ext}} = 0 \tag{24}$$

where $\phi_t$ is a temporary signed distance implicit function and $V_{\text{ext}}$ is the extended velocity function. The extended velocity function is constrained to maintain the values already computed along the boundary.

## Gradient computation

An accurate estimation of the gradient $\nabla\phi$ is essential in solving Hamilton-Jacobi equation using upwind scheme. A current gradient computation evoked internally during level-set update utilizes 5th order Hamilton-Jacobi WENO (weighted essentially non-oscillatory) method. It gives out a better approximation when compared with ENO.

Spatial derivatives of $\phi$ and spatial stencils are obtained depending on the sign of the velocity and the location of the node in the domain of interest. In any case, five stencil values are obtained with same distances $(v_1, v_2, v_3, v_4, v_5)$. Based on the smoothness of each stencil, we calculate normalized weights $(w_1, w_2, w_3)$ whose sum is equal to unity.

$$\nabla\phi = w_1(2v_1 - 7v_2 + 11v_3) + w_2(5v_3 - v_2 + 2v_4) + w_3(2v_3 + 5v_4 - v_5) \tag{25}$$

Detailed numerical steps can be found in the reference [OF03]

## Convergence criterion

The convergence criterion is computed if the volume constraint is satisfied and is defined using the maximum change in compliance over the previous 5 iterations:

$$\Delta C^k = \max\left(|C^k - C^m|/C^k\right), \quad m \in [k-5, k-1] \tag{26}$$

where $C^k$ is the compliance computed at iteration $k$ and the optimization process is terminated if $\Delta C^k < 1e-3$.

# Bibliography

[AJT04] Grégoire Allaire, François Jouve, and Anca-Maria Toader. Structural optimization using sensitivity analysis and a level-set method. *Journal of Computational Physics*, 194(1):363–393, 2004.

[AS99] D Adalsteinsson and J. A Sethian. The fast construction of extension velocities in level set methods. *Journal of Computational Physics*, 148(1):2–22, 1999.

[DK15] Peter D. Dunning and H. Alicia Kim. Introducing the sequential linear programming level-set method for topology optimization. *Structural and Multidisciplinary Optimization*, 51(3):631–643, 2015.

[OF03] Stanley Osher and Ronald Fedkiw. *Level set methods and dynamic implicit surfaces*, volume 153 of *Applied Mathematical Sciences*. Springer, London, 2003.

[SDK16] Raghavendra Sivapuram, Peter D. Dunning, and H. Alicia Kim. Simultaneous material and structural optimization by multiscale topology optimization. *Structural and Multidisciplinary Optimization*, pages 1–15, 2016.

[WW06] Shengyin Wang and Michael Y. Wang. A moving superimposed finite element method for structural topology optimization. *International Journal for Numerical Methods in Engineering*, 65(11):1892–1922, 2006.