Tutorial for OpenLSTO v0.1:

Open Source Level Set Topology Optimization

M2DO Lab[1,2]

[1]Cardiff University

[2]University of California, San Diego

November 2017

Users' Guide

Software components

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.

- **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

OpenLSTO 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 Github

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://github.com/M2DOLab/OpenLSTO.git

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.

http://m2do.ucsd.edu/static/zip/OpenLSTO-v0.1.zip

Installation

OpenLSTO 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 OpenLSTO can be extended using the externally-provided software. Again, to facilitate ease of use and to promote the open source nature, whenever external software is required within the OpenLSTO 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 OpenLSTO 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 OpenLSTO 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 OpenLSTO 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, OpenLSTO 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

OpenLSTO 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), OpenLSTO 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 OpenLSTO. Consequently, the following capabilities of OpenLSTO 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 OpenLSTO. 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.

Tutorial

The following tutorial will walk you through the steps required to solve a compliance minimization problem using OpenLSTO. It is assumed that you have already obtained the OpenLSTO code. If you have not, please see the Download and Installation pages.

Nomenclature

Dirichlet boundary conditions:	the value of the function on a surface.
Neumann boundary conditions:	the normal derivative of the function on a surface.
Euclidean distance:	the straight-line distance between two points in Euclidean space.
Signed distance function:	the distance of a given point $\boldsymbol{x}$ from the boundary of Ω, with the sign determined by whether $\boldsymbol{x}$ is in Ω.
Level set function:	a set where the function, e.g. signed distance, takes on a given constant value c, e.g. 0 for boundary of a shape.
CFL condition:	the Courant–Friedrichs–Lewy (CFL) condition is a necessary condition for convergence while solving certain partial differential equations (usually hyperbolic PDEs) numerically by the method of finite differences
Marching square algorithm:	a computer graphics algorithm that generates contours for a two-dimensional scalar field.
Upwind finite difference:	a class of numerical discretization methods using an adaptive or solution-sensitive finite difference stencil to numerically simulate the direction of propagation of information for solving hyperbolic partial differential equations.

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 five parts: (1) settings for the finite element analysis, (2) settings for the sensitivity analysis, (3) settings for the level set method, (4) settings for the optimization and (5) the level set topology optimization loop. Details for each are explained below.

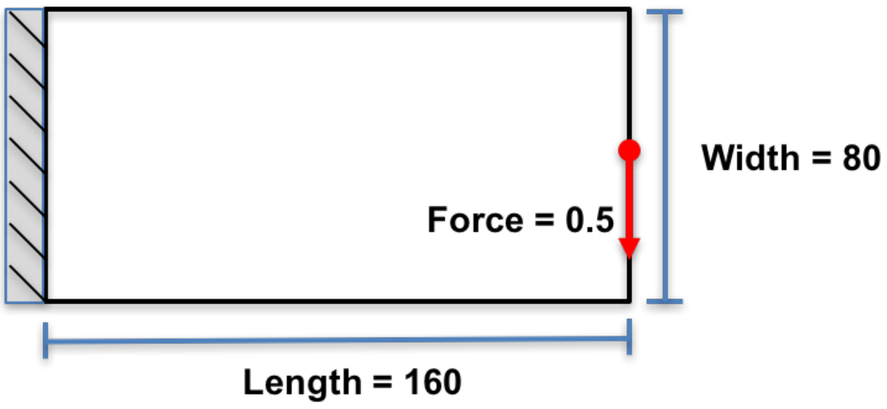


Figure 1: Configuration of the cantilever beam

Settings for the Finite Element Analysis - lines 33 - 162

The optimization domain is assumed to be rectangular and split into square finite elements with unit width and height. 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.

A two dimensional `M2DO_FEA::Mesh` class, which will hold information pertaining to nodes, elements and degrees of freedom, is instantiated at line 44. The rectangular design domain is defined by its four corner points in the `fea_box` parameter at lines 52 - 55, and the structured mesh is generated using the `MeshSolidHyperRectangle` function at line 61. The degrees of freedom are assigned using `AssignDof` at line 57. Material properties: Young's modulus, Poisson's ratio and density are added to the mesh at lines 69 - 73.

Line 80 defines an instance of an `M2DO_FEA::StationaryStudy` class, which is capable of solving problems of the form $[K]\{u\} = \{f\}$. Dirichlet boundary conditions are defined in lines 83 - 114 in order to fix the degrees of freedom on the left-most edge. A point load is defined on the mid right-most edge in lines 117 - 147. Since, in this example, the load vector $\{f\}$ is design-independent, it is built at this point using the `AssembleF` function.

Lastly, the FEA solver settings: initial solution guess and convergence tolerance, are defined at lines 153 - 157.

Settings for the Sensitivity Analysis - lines 162 - 170

The `M2DO_FEA::Sensitivity` class is used to compute the sensitivity of compliance with respect to movements of the structural boundary. In this example, it suffices to declare one instance, which is done in line 168.

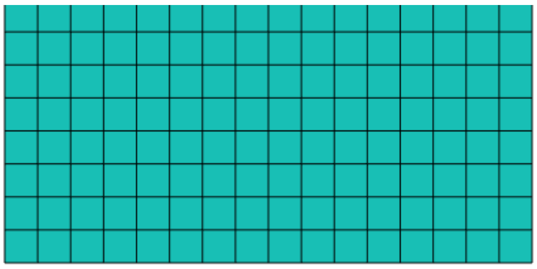


Figure 2: FEA and LSM meshing of design domain

Settings for the Level Set Method - lines 173 - 228

Basic parameters related to level set method are defined first. Line 183 sets the `move_limit`: the boundary will move less than this between design iterations. The `band_width` parameter defines the width of the narrow band: the area nearest the structural boundary which is re-initialized between iterations. Several holes are then initialized in the level set function at lines 193 - 226.

Settings for the Optimization - lines 231 - 255

The `max_time` and `max_iterations` parameters, set at lines 241 - 242, limit the running time and number of design iterations respectively. Finite elements with area fractions less than `max_area`, set at line 243, are ignored in the sensitivity analysis calculations. The `max_diff` parameter at line 244 specifies the change in objective function required to signify convergence.

Level Set Topology Optimization Loop - lines 258 - 463

An instance of an `M2DO_LSM::Mesh` class is created at line 270, and is given the same resolution as the FEA mesh above. Hence, a level-set array of `(nelx+1)*(nely+1)` grid points are created. For each, the level set function is calculated as the Euclidean distance to the nearest structural boundary using the `reinitialize` function. For nodes on solid elements the sign is chosen to be negative, whereas for nodes on void elements the sign is chosen to be positive. The boundary of the structure is to be stored in a `M2DO_LSM::Boundary` object, instantiated at line 219.

In the next part of program, the optimization loop is carried out, with a convergence check included to terminate the iteration when satisfactory solution is obtained. Iterations are counted with `n_iterations` and continue for a maximum of `max_it`, 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 (line 387).

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 a default. The `n_reinit` variable 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 1×10^{-3} comparing with the current objective value and the volume is within 1×10^{-4} of the required value `max_area` (lines 469 - 471).

Outputs

The code allows to yield 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

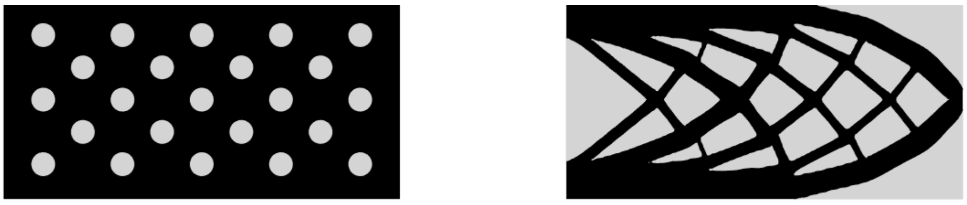

(a) Initial design (b) Final solution

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

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 activate 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.

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

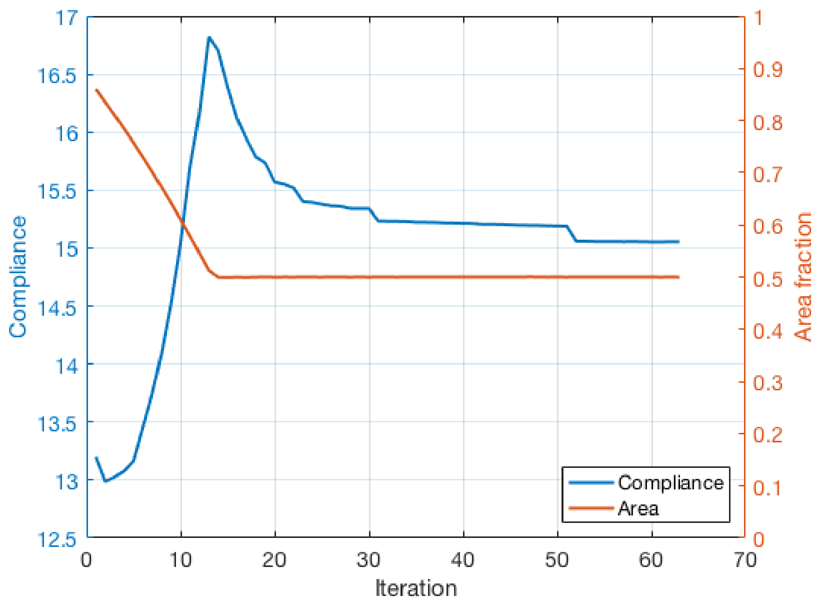


Figure 4: Convergence history of compliance value and area fraction for the cantilever example

half of the beam needs to be solved due to the 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.

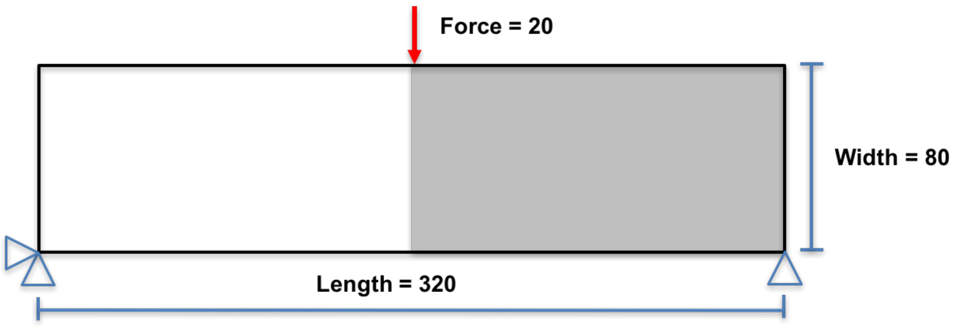
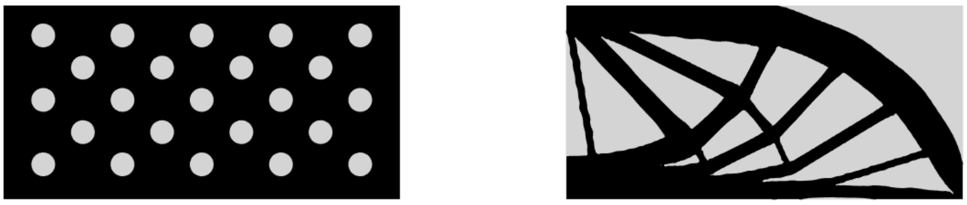


Figure 5: Configuration of the MBB beam


(a) Initial design

(b) Final solution

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

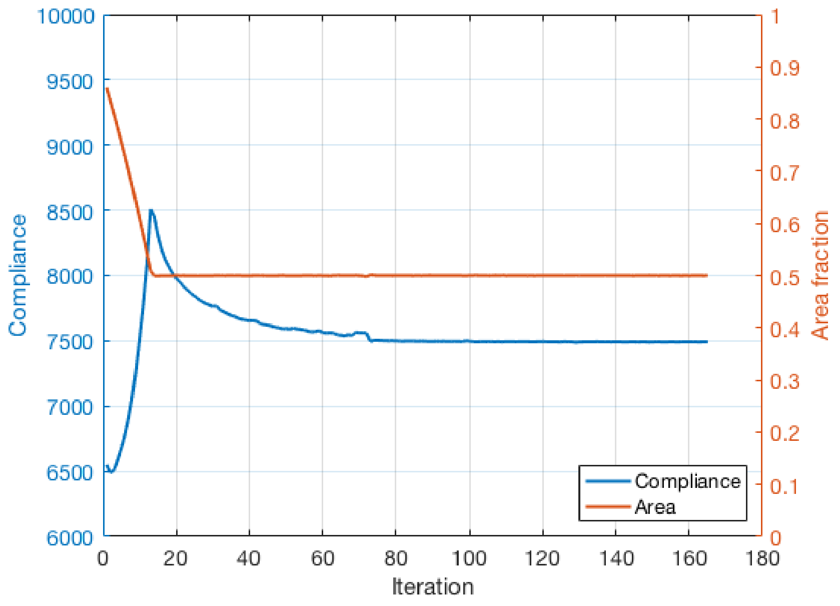


Figure 7: Convergence history of compliance value and area fraction for the MBB example

C++ code for compliance minimization of a cantilever beam

```cpp
#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 () {

    ////////////////////////////////////////////////////////////////////////
    //                       //
    //    Part 1: SETTING FOR FINITE ELEMENT ANALYSIS            //
    //                       //
    ////////////////////////////////////////////////////////////////////////

    /*
     Dimensionality of problem:
     */

    const int spacedim = 2 ;

    /*
     FEA & level set mesh parameters:
     */

    const unsigned int nelx = 160, nely = 80;

    /*
     Create an FEA mesh object.
     */

    FEA::Mesh fea_mesh (spacedim) ;

    /*
     Mesh a hyper rectangle.
     */
    Matrix<double,-1,-1> fea_box (4, 2) ;

    // define design domain
    fea_box.data = {{0,0},{nelx,0},{nelx,nely},{0,nely}};

    // mesh size in horizontal and vertical directions
    vector<int> nel = {nelx, nely} ;

    int element_order = 2 ;
    fea_mesh.MeshSolidHyperRectangle (nel, fea_box, element_order, false) ;
    fea_mesh.is_structured = true ;
    fea_mesh.AssignDof () ;

    /*
     Add material properties:
     */

    double E = 1.0, v = 0.3, rho = 1.0;
    fea_mesh.solid_materials.push_back (FEA::SolidMaterial (spacedim, E, v, rho)) ;

    /*
     Next we specify that we will undertake a stationary study, which takes the form [K]{u} = {f}.
     */

    FEA::StationaryStudy fea_study (fea_mesh) ;

    /*
     Add a homogeneous Dirichlet boundary condition (fix some nodes).
     */

    // Example 1: cantilever beam

    // vector<int> fixed_nodes = fea_mesh.GetNodesByCoordinates ({0.0, 0.0}, {0.1, 1.0E10}) ;
    // vector<int> fixed_dof = fea_mesh.dof (fixed_nodes) ;

    // Example 2: half of simply supported beam or Messerschmitt-Bolkow-Blohm (MBB) beam

    vector<int> fixed_nodes_left = fea_mesh.GetNodesByCoordinates ({0.0, 0.0}, {0.1, 1.0E10}) ;
    vector<int> fixed_dof_left = fea_mesh.dof (fixed_nodes_left, {0}) ;

    vector<int> fixed_nodes_right = fea_mesh.GetNodesByCoordinates ({nelx, 0.0}, {0.1, 0.1}) ;
    vector<int> fixed_dof_right = fea_mesh.dof (fixed_nodes_right, {1}) ;

    vector<int> fixed_dof;
    fixed_dof.reserve( fixed_dof_left.size() + fixed_dof_right.size() );
    fixed_dof.insert( fixed_dof.end(), fixed_dof_left.begin(), fixed_dof_left.end() );
    fixed_dof.insert( fixed_dof.end(), fixed_dof_right.begin(), fixed_dof_right.end() );

    fea_study.AddBoundaryConditions (FEA::HomogeneousDirichletBoundaryConditions (fixed_dof, fea_mesh.n_dof)) ;

```

```
/*
 Apply a point load.
 */

// Example 1: cantilever beam

// vector<int> load_node = fea_mesh.GetNodesByCoordinates ({1.0*nelx, 0.5*nely}, {1e-12, 1e-12}) ;
// 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_val[2*i+1] = -0.5 ;

// }

// Example 2: half of simply supported beam or Messerschmitt-Bolkow-Blohm (MBB) beam

vector<int> load_node = fea_mesh.GetNodesByCoordinates ({0, nely}, {1e-12, 1e-12}) ;
vector<int> load_dof  = fea_mesh.dof (load_node, {1}) ;
vector<double> load_val (load_node.size()) ;

for (int i = 0 ; i < load_node.size() ; ++i) {

    load_val[i] = -10.0 ;

}

FEA::PointValues point_load (load_dof, load_val) ;
fea_study.AssembleF (point_load, false) ;

// Create sensitivity analysis instance.
FEA::SensitivityAnalysis sens(fea_study) ;

////////////////////////////////////////////////////////////////////////
//                       //
//     Part 2: SETTING FOR LEVEL SET METHOD         //
//                       //
////////////////////////////////////////////////////////////////////////

/*
 Tread carefully; these be Renato's additions (very funny!).
 */

// Maximum displacement per iteration, in units of the mesh spacing.
// This is the CFL limit.
double moveLimit = 0.5 ;

// Set maximum running time.
double maxTime = 6000 ;
int maxit = 300;

// Set sampling interval.
double sampleInterval = 50 ;

```

```
    // Set time of the next sample.
    double nextSample = 50 ;

    // Maximum material area.
    double maxArea = 0.5 ;

    // Default temperature of the thermal bath.
    double temperature = 0 ;

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

    double meshArea = lsmMesh.width * lsmMesh.height ;

    // Create two horizontal rows with four equally space holes.
    vector<LSM::Hole> holes ;

    holes.push_back(LSM::Hole(16, 14, 5)) ;
    holes.push_back(LSM::Hole(32, 27, 5)) ;
    holes.push_back(LSM::Hole(48, 14, 5)) ;
    holes.push_back(LSM::Hole(64, 27, 5)) ;
    holes.push_back(LSM::Hole(80, 14, 5)) ;
    holes.push_back(LSM::Hole(96, 27, 5)) ;
    holes.push_back(LSM::Hole(112, 14, 5)) ;
    holes.push_back(LSM::Hole(128, 27, 5)) ;
    holes.push_back(LSM::Hole(144, 14, 5)) ;

    holes.push_back(LSM::Hole(16, 40, 5)) ;
    holes.push_back(LSM::Hole(32, 53, 5)) ;
    holes.push_back(LSM::Hole(48, 40, 5)) ;
    holes.push_back(LSM::Hole(64, 53, 5)) ;
    holes.push_back(LSM::Hole(80, 40, 5)) ;
    holes.push_back(LSM::Hole(96, 53, 5)) ;
    holes.push_back(LSM::Hole(112, 40, 5)) ;
    holes.push_back(LSM::Hole(128, 53, 5)) ;
    holes.push_back(LSM::Hole(144, 40, 5)) ;

    holes.push_back(LSM::Hole(16, 66, 5)) ;
    holes.push_back(LSM::Hole(48, 66, 5)) ;
    holes.push_back(LSM::Hole(80, 66, 5)) ;
    holes.push_back(LSM::Hole(112, 66, 5)) ;
    holes.push_back(LSM::Hole(144, 66, 5)) ;

    // Initialise guess solution for cg
    int n_dof = fea_mesh.n_dof ;
    std::vector<double> u_guess(n_dof,0.0);

    // Initialise the level set object (from the hole vector).
    LSM::LevelSet levelSet(lsmMesh, holes, moveLimit, 6, false) ;

    // Initialise io object.
    LSM::InputOutput io ;

    // Reinitialise the level set to a signed distance function.
    levelSet.reinitialise() ;

    // Initialise the boundary object.
    LSM::Boundary boundary(levelSet) ;

    // Initialise random number generator.
    LSM::MersenneTwister rng ;

    // Number of cycles since signed distance reinitialisation.
    unsigned int nReinit = 0 ;

    // Running time.
    double time = 0 ;

    // Time measurements.
    vector<double> times ;

    // Compliance measurements.
    vector<double> compliances ;

    // Boundary curvature measurements.
    vector<double> areas ;

    /* Lambda values for the optimiser.
     These are reused, i.e. the solution from the current iteration is
     used as an estimate for the next, hence we declare the vector
     outside of the main loop.
     */
    vector<double> lambdas(2) ;

    ///////////////////////////////////////////////////////////////////////
    //                      //
    //   Part 3: ITERATION FOR LEVEL SET TOPOLOGY OPTIMIZATION      //
    //                      //
    ///////////////////////////////////////////////////////////////////////

    // setup for outputing convergence history of objective function values and
        constraints
    ofstream convergenceHistory;
    convergenceHistory.open("convergenceHistory.txt",ofstream::out);

    // write initial design into vtk & txt format - signed distance
    std::ostringstream fileNameLSF, fileNameArea;
    fileNameLSF.str("");
    fileNameLSF << "level-set-initial.vtk";
    io.saveLevelSetVTK(fileNameLSF, levelSet);
    fileNameLSF.str("");
    fileNameLSF << "level-set-initial.txt";
    io.saveLevelSetTXT(fileNameLSF, levelSet, true);

    // write optimal design into vtk & txt format - area fraction
    fileNameArea.str("");
    fileNameArea << "area-initial.vtk";
    io.saveAreaFractionsVTK(fileNameArea, lsmMesh);
    fileNameArea.str("");
    fileNameArea << "area-initial.txt";
    io.saveAreaFractionsTXT(fileNameArea, lsmMesh);

    // flag for chosing whether write level set function and area fraction into vtk
         files or not
    bool flagOutputHistory = false;

    // Integrate until we exceed the maximum time.
    int n_iterations = 0 ;

    // Initialise vector to save objective history.
    std::vector<double> Objective_Values;
    // Initialise variable to stop loop.
    double Relative_Difference = 1.0;

    // iterative calculation

    cout << "\nStarting compliance minimisation demo...\n\n" ;

    while (n_iterations < maxit) {

        ++n_iterations ;
        //cout << "Echo 1" << endl;

        // Perform boundary discretisation.
        boundary.discretise(false, lambdas.size()) ;

        // Compute element area fractions.
        boundary.computeAreaFractions() ;

        // Assign area fractions.
        for (unsigned int i=0 ; i< fea_mesh.solid_elements.size() ; i++) {

            if (lsmMesh.elements[i].area < 1e-3) {

                fea_mesh.solid_elements[i].area_fraction = 1e-3 ;

            }
            else {

                fea_mesh.solid_elements[i].area_fraction = lsmMesh.elements[i].area
                    ;

            }

        }

        /*
         Assemble stiffness matrix [K] using area fraction method:
         */
        fea_study.Assemble_K_With_Area_Fractions_Sparse (false) ;

        /*
         Solve equation:
         */

        double cg_tolerence = 1.0e-6;
        fea_study.Solve_With_CG (false, cg_tolerence,u_guess) ;

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

        double abs_bsens_max = 0.0;
        for (int i=0 ; i<boundary.points.size() ; i++) {

            vector<double> boundary_point (2, 0.0) ;
            boundary_point[0] = boundary.points[i].coord.x;
            boundary_point[1] = boundary.points[i].coord.y;

            // Interpolate Guass point sensitivities by least squares.
            sens.ComputeBoundarySensitivities(boundary_point) ;


            // Assign sensitivities.
            boundary.points[i].sensitivities[0] = -sens.boundary_sensitivities[i] ;
            boundary.points[i].sensitivities[1] = -1 ;

            abs_bsens_max = std::max(abs_bsens_max, std::abs(sens.
                boundary_sensitivities[i]));
        }

        // clearing sens.boundarysens vector.
        sens.boundary_sensitivities.clear() ;

        // Time step associated with the iteration.
        double timeStep ;

        // Constraint distance vector.
        vector<double> constraintDistances ;

        // Push current distance from constraint violation into vector.
        constraintDistances.push_back(meshArea*maxArea - boundary.area) ;

        /* Initialise the optimisation object.

         The Optimise class is a lightweight object so there is no cost for
         reinitialising at every iteration. A smart compiler will optimise
         this anyway, i.e. the same memory space will be reused. It is better
         to place objects in the correct scope in order to aid readability
         and to avoid unintended name clashes, etc.
         */
        LSM::Optimise optimise(boundary.points,  timeStep, moveLimit) ;

        // set up required parameters
        optimise.length_x = lsmMesh.width;
        optimise.length_y = lsmMesh.height;
        optimise.boundary_area = boundary.area; // area of structure
        optimise.mesh_area = meshArea; // area of the entire mesh
        optimise.max_area = maxArea; // maximum area

        // Perform the optimisation.
        optimise.Solve_With_NewtonRaphson() ;

        // Extend boundary point velocities to all narrow band nodes.
        levelSet.computeVelocities(boundary.points, timeStep, temperature, rng) ;

        // Compute gradient of the signed distance function within the narrow band.
        levelSet.computeGradients() ;

        // Update the level set function.
        bool isReinitialised = levelSet.update(timeStep) ;

        // Reinitialise the signed distance function, if necessary.
        if (!isReinitialised) {

            // Reinitialise at least every 20 iterations.
            if (nReinit == 20) {

                levelSet.reinitialise() ;
                nReinit = 0 ;
            }

        }
        else nReinit = 0 ;

        // Increment the number of steps since reinitialisation.
        nReinit++ ;

        // Increment the time.
        time += timeStep;

        // Calculate current area fraction.
        double area = boundary.area / meshArea ;

        // Record the time, compliance, and area.
        times.push_back(time) ;
        //compliances.push_back(study.compliance);
        areas.push_back(area) ;
        Objective_Values.push_back(sens.objective);

        // Write objective function values and area into txt file
        convergenceHistory << n_iterations << "\t" << setprecision(15) <<
            Objective_Values[n_iterations-1] << "\t" << area << endl;

        // Converence criterion [Dunning_11_FINEL]
        double Objective_Value_k, Objective_Value_m;
        if (n_iterations > 5) {

            Objective_Value_k = sens.objective;
            Relative_Difference = 0.0;
            for (int i = 1; i <= 5; i++) {

                Objective_Value_m = Objective_Values[n_iterations - i - 1];
                Relative_Difference = max(Relative_Difference, abs((
                    Objective_Value_k - Objective_Value_m)/Objective_Value_k));

            }

        }

        if (n_iterations==1) {

            // Print output header.
            printf("--------------------------------\n");
            printf("%8s %12s %10s\n", "Iteration", "Compliance", "Area");
            printf("--------------------------------\n");

        }
        // Print statistics.
        printf("%8.1f %12.4f %10.4f\n", double (n_iterations), sens.objective, area
            ) ;

        // Write level set and boundary segments to file.
        if (flagOutputHistory) {

            io.saveLevelSetVTK(n_iterations, levelSet) ;
            io.saveAreaFractionsVTK(n_iterations, lsmMesh) ;
        }

        // Write optimal design into vtk & txt format - area fraction
        fileNameArea.str("");
        fileNameArea << "area-optimal.vtk";
        io.saveAreaFractionsVTK(fileNameArea, lsmMesh);
        fileNameArea.str("");
        fileNameArea << "area-optimal.txt";
        io.saveAreaFractionsTXT(fileNameArea, lsmMesh);

        // Write optimal design into vtk & txt format - signed distance
        fileNameLSF.str("");
        fileNameLSF << "level-set-optimal.vtk";
        io.saveLevelSetVTK(fileNameLSF, levelSet);
        fileNameLSF.str("");
        fileNameLSF << "level-set-optimal.txt";
        io.saveLevelSetTXT(fileNameLSF, levelSet, true);

        if ((Relative_Difference < 0.0001) & (area < 1.001*maxArea)) {
            break;
        }
    }

    /*
     Aaaaaand that's all, folks!
     */

    cout << "\nProgram complete.\n\n" ;

    return 0 ;
}
```