Tutorial 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

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 and 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 Apache V. 2.0 license. 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 it's github page directly. You are able to download the code repository as a ZIP file from the github page.

OpenLSTO has been designed with ease of installation and use in mind. This means that, whenever 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. Capabilities of OpenLSTO can also be extended using the externally-provided software. Again, to facilitate ease of use and to promote the open source nature, no matter when 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 users, the native terminal applications are needed.

For Windows users, downloading and installing MinGW (http://www.mingw.org/) is a prerequisite to establish Unix/Linux-like development environment. Be sure to check GNU Make tool to be installed. As unix-like shell commands are used during compilation and execution of the program, additional msys tools such as mkdir is highly recommended.

Data Visualisation

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 for Windows, Unix/Linux and Mac OS 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. Windows or Mac Users: For compatibility, the OpenMP flag *-fopenmp* in the second line of the makefile (starts with *CFLAGS*) should be removed before compilation.

Note that compilation and execution need to be done in the specific module folder, e.g. *projects/compliance* for the compliance problem.

Compile:

»make main

Run a simulation and output results:

»./bin/a.out (for Windows users, this command should be changed to *start ./bin/a.out*)

Clean existing compiled files:

»make clean

clean existing object files:

»make clean_obj

clean existing output files:

»make clean_results

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.

Version History

- v0.1: 2017.11.01, M2DO at Cardiff University & University of California, San Diego.
- v1.0: 2018.08.27, M2DO at Cardiff University & University of California, San Diego.

Tutorial 1: Compliance Minimization Problem for 2D

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 solutions from other topology optimization approaches, such as solid isotropic material with penalization (SIMP) and bi-directional evolutionary structural optimization (BESO), as a validation case for OpenLSTO. Consequently, the following capabilities of OpenLSTO will be showcased in this tutorial:

- finite element analysis of a structure using an 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** in 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 not, please refer to Download.

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 or not
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;

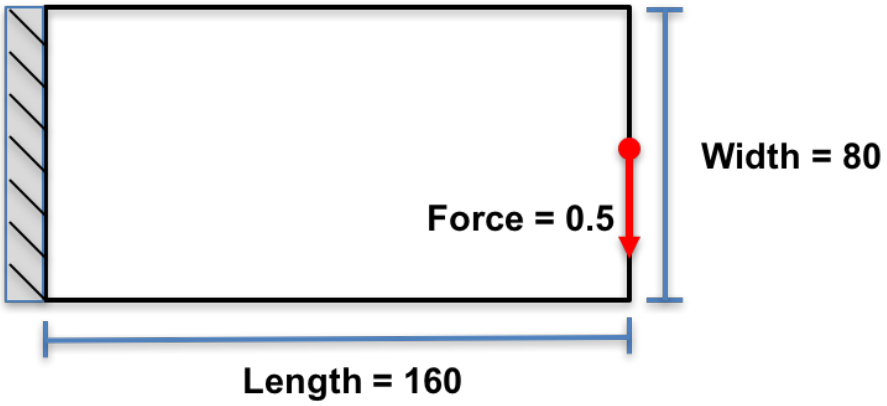


Figure 1: Configuration of the cantilever beam.

3. settings for the level set method;

4. settings for the optimization;

5. the level set topology optimization loop.

Details for each part are explained below.

Settings for the Finite Element Analysis

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 Figure 2 shows.

First, a finite element model is created through the following lines:

```
/*
 FEA Mesh:
*/

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

// Number of elements in x and y directions:
const unsigned int nelx = 160, nely = 80 ;

// fea_box contains the (x,y) coordinates of 4 corner points of rectangle
    containing the mesh:
MatrixXd fea_box(4,2);

fea_box <<   0.0,  0.0,
   nelx,  0.0,
   nelx, nely,
    0.0, nely;

// Element Gauss integration order:
```

```
20     int element_order = 2 ;
21
22     // Create structured mesh and assign degrees of freedom:
23   fea_mesh.MeshSolidHyperRectangle ({nelx, nely}, fea_box, element_order, false) ;
24   fea_mesh.is_structured = true ;
25   fea_mesh.AssignDof () ;
```

A two dimensional `FEA::Mesh` class, which will hold information pertaining to nodes, elements and degrees of freedom, is instantiated. A rectangular design domain is defined by its four corner points in the `fea_box` parameter, and the structured mesh is generated using the `MeshSolidHyperRectangle` function. The degrees of freedom are assigned using `AssignDof`.

Material properties such as Young's modulus, Poisson's ratio and density are added to the mesh as follows:

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

An instance of the `FEA::StationaryStudy` class is created, which is capable of solving problems of the form $[K]\{u\} = \{f\}$. Dirichlet boundary conditions are defined in order to fix the degrees of freedom on the left-most edge. A point load is defined on the mid right-most edge. Since in this example the load vector $\{f\}$ is design-independent, the point load is built using the `AssembleF` function. These are realized in the following lines:

```
1   /*
2    Next we specify that we will undertake a stationary study, which takes the
3    form [K]{u} = {f}:
4   */
5
6    FEA::StationaryStudy fea_study (fea_mesh) ;
7
8   /*
9    Define homogeneous Dirichlet boundary condition (fixed nodes) and add to study:
10  */
11
12  // Select dof using a box centered at coord of size tol:
13  vector<double>    coord = {0.0, 0.0}, tol = {1e-12, 1e10} ;
14  vector<int> fixed_nodes = fea_mesh.GetNodesByCoordinates (coord, tol) ;
15  vector<int>   fixed_dof = fea_mesh.dof (fixed_nodes) ;
16
17     // Add boundary conditions to study:
18  vector<double> amplitude (fixed_dof.size(),0.0) ; // Values equal to zero.
19     fea_study.AddBoundaryConditions (FEA::DirichletBoundaryConditions (fixed_dof,
           amplitude, fea_mesh.n_dof)) ;
20
21  /*
22   Define a point load of (0, -0.5) at the point (nelx, 0.5*nely) and add to study:
23  */
24
25   Select dof using a box centered at coord of size tol:
26   coord = {1.0*nelx, 0.5*nely}, tol = {1e-12, 1e-12} ;
27   vector<int> load_node = fea_mesh.GetNodesByCoordinates (coord, tol) ;
28   vector<int>  load_dof = fea_mesh.dof (load_node) ;
```

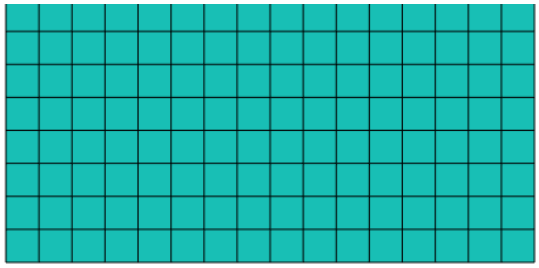


Figure 2: Overlapped FEA and LSM meshes of the design domain.

```
29
30    vector<double> load_val (load_node.size() * 2) ;
31    for (int i = 0 ; i < load_node.size() ; ++i) {
32     load_val[2*i]   = 0.00 ; // load component in x direction.
33     load_val[2*i+1] = -0.5 ; // load component in y direction.
34   }
35
36   // Add point load to study and assemble load vector {f}:
37   FEA::PointValues point_load (load_dof, load_val) ;
38   fea_study.AssembleF (point_load, false) ;
```

Lastly, the FEA solver settings (initial solution guess and convergence tolerance) are defined as:

```
1   // Initialise guess solution for CG:
2   vector<double> u_guess (fea_mesh.n_dof, 0.0) ;
3
4   // Convergence tolerance:
5   double cg_tolerence = 1.0e-6 ;
```

Settings for the Sensitivity Analysis

The `FEA::SensitivityAnalysis` 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 as:

```
1   FEA::SensitivityAnalysis sens (fea_study) ;
```

Settings for the Level Set Method

Basic parameters width related to level set method are defined first. They include `move_limit` defining that the level set boundary will move less than this value between design iterations, and `band_width` for the width of the narrow band: the area nearest the structural boundary which is re-initialized between iterations.

```
1   /*
2    Define LSM parameters:
3   */
4
5   double     move_limit = 0.5 ;   // Maximum displacement per iteration in units of
        the mesh spacing.
```

```
6   double    band_width = 6 ;     // Width of the narrow band.
7   bool is_fixed_domain = false ; // Whether or not the domain boundary is fixed.
```

Several holes are then seeded in the level set function. An vector array of hole objects are declared first, and the `LSM::Hole` objects are appended. A constructor of the object is declared as (x,y,r): location of x, location of y, and a radius of the hole.

```
1   /*
2    Seed initial holes:
3    In this example, we create five horizontal rows, each row alternating between
4    four and five equally spaced holes, all of radius 5 units.
5   */
6
7   vector<LSM::Hole> holes ;
8
9   // First row with five holes:
10  holes.push_back (LSM::Hole (16, 14, 5)) ;
11  holes.push_back (LSM::Hole (48, 14, 5)) ;
12  holes.push_back (LSM::Hole (80, 14, 5)) ;
13  holes.push_back (LSM::Hole (112, 14, 5)) ;
14  holes.push_back (LSM::Hole (144, 14, 5)) ;
15
16  // Second row with four holes:
17  holes.push_back (LSM::Hole (32, 27, 5)) ;
18  holes.push_back (LSM::Hole (64, 27, 5)) ;
19  holes.push_back (LSM::Hole (96, 27, 5)) ;
20  holes.push_back (LSM::Hole (128, 27, 5)) ;
21
22  // Third row with five holes:
23  holes.push_back (LSM::Hole (16, 40, 5)) ;
24  holes.push_back (LSM::Hole (48, 40, 5)) ;
25  holes.push_back (LSM::Hole (80, 40, 5)) ;
26  holes.push_back (LSM::Hole (112, 40, 5)) ;
27  holes.push_back (LSM::Hole (144, 40, 5)) ;
28
29  // Fourth row with four holes:
30  holes.push_back (LSM::Hole (32, 53, 5)) ;
31  holes.push_back (LSM::Hole (64, 53, 5)) ;
32  holes.push_back (LSM::Hole (96, 53, 5)) ;
33  holes.push_back (LSM::Hole (128, 53, 5)) ;
34
35  // Fifth row with five holes:
36  holes.push_back (LSM::Hole (16, 66, 5)) ;
37  holes.push_back (LSM::Hole (48, 66, 5)) ;
38  holes.push_back (LSM::Hole (80, 66, 5)) ;
39  holes.push_back (LSM::Hole (112, 66, 5)) ;
40  holes.push_back (LSM::Hole (144, 66, 5)) ;
```

Settings for the Optimization

Before solving the optimization problem, several parameters are set. `max_iterations` limits the number of design iterations; `max_area` defines the constraint of finite elements with area fractions, but it is ignored in the sensitivity analysis calculations;

`max_diff` specifies the change in objective function required to signify convergence; `lambdas` are used to store weighting factors.

```
/*
 Define parameters needed for optimization loop:
*/

int    max_iterations = 300 ;    // maximum number of iterations.
double       max_area = 0.5 ;    // maximum material area.
double       max_diff = 0.0001 ; // relative difference between iterations must be
     less than this value to reach convergence.

/*
 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) ;
```

Level Set Topology Optimization Loop

An instance of an `LSM::Mesh` class is created 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 `LSM::Boundary` object.

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

double mesh_area = lsm_mesh.width * lsm_mesh.height ;

// 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 () ;

// Initialise the boundary object :
LSM::Boundary boundary (level_set) ;
```

```
/*
 Optimization:
*/

// Declare parameters that will change within the optimization loop:
```

```
6   unsigned int   n_reinit = 0 ;              // num cycles since signed dist
        reinitialisation.
7   double         time     = 0 ;              // running time.
8   vector<double> times, compliances, areas ; // time, compliance and area
        measurements.
9   int            n_iterations = 0 ;          // iteration counter
10  vector<double> objective_values ;          // vector to save objective history
11  double         relative_difference = 1.0 ; // convergence criteria variable,
12
13  // Initialise io object:
14  LSM::InputOutput io ;
15
16  cout << "\nStarting compliance minimisation demo...\n\n" ;
17
18  // Print output header:
19  printf ("---------------------------------\n") ;
20  printf ("%8s %12s %10s\n", "Iteration", "Compliance", "Area") ;
21  printf ("---------------------------------\n") ;
22
23  // Create directories for output if the don't already exist
24  system("mkdir -p results/history");
25  system("mkdir -p results/level_set");
26  system("mkdir -p results/area_fractions");
27  system("mkdir -p results/boundary_segments");
28
29  // Remove any existing files from output directories
30  system("find ./results -type f -name '*.txt' -delete");
31  system("find ./results -type f -name '*.vtk' -delete");
32
33  // Setup text file:
34  ofstream history_file ;
35  history_file.open ("results/history/history.txt", ios_base::app) ;
36  history_file << "Iteration\tCompliance\tArea\n" ;
37  history_file.close () ;
38
39
40
41  // END OF LEVEL SET TOPOLOGY OPTIMIZATION LOOP
```

In the next part of the program, the optimization loop that starts, and it is terminated with a convergence check when satisfactory solution is obtained. Iterations are counted by variable `n_iterations` and continue for a maximum of `max_iterations`. Inside the optimization loop, the boundary of the structure defined by iso-contour (2D) or surface (3D), e.g. zero level set, is discretized by finding intersection points on mesh grid with the use of marching square algorithm. The area fraction of each level set element is then calculated. The area fractions of elements are assigned to the finite elements. The area fraction fixed grid finite element method is called to conduct finite element analysis through assembling global stiffness matrix and solving finite element equation. The shape sensitivity of the compliance at each gauss points in finite element is calculated. Shape sensitivities are then calculated for boundary points by extrapolating or interpolating from the information at gauss points with the use of the weighted least square method. The volume constraint is also imposed

at each iteration. By completing these necessary inputs, the Lagrangian Multiplier method is applied to solve the optimization problem.

```
while (n_iterations < max_iterations) {

 ++n_iterations ;

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

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

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

  if (lsm_mesh.elements[i].area < 1e-3) {
   fea_mesh.solid_elements[i].area_fraction = 1e-3 ;
  }

  else {
   fea_mesh.solid_elements[i].area_fraction = lsm_mesh.elements[i].area ;
  }

 }

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

 // Solve equation using conjugant gradient (cg) method:
 fea_study.SolveWithCG();

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

 // Compute compliance sensitivities at boundary points:
 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 Gauss 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;

 }

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

 // Time step associated with the iteration
 double time_step ;

 // Constraint distance vector
 vector<double> constraint_distances ;
```

```
57
58    // Push current distance from constraint violation into vector
59    constraint_distances.push_back (mesh_area * max_area - boundary.area) ;
60
61    /* Initialise the optimisation object
62
63    The Optimise class is a lightweight object so there is no cost for
64    reinitialising at every iteration. A smart compiler will optimise
65    this anyway, i.e. the same memory space will be reused. It is better
66    to place objects in the correct scope in order to aid readability
67    and to avoid unintended name clashes, etc.
68    */
69
70    LSM::Optimise optimise (boundary.points, time_step, move_limit) ;
71
72    // set up required parameters
73    optimise.length_x = lsm_mesh.width ;
74    optimise.length_y = lsm_mesh.height ;
75    optimise.boundary_area = boundary.area ; // area of structure
76    optimise.mesh_area = mesh_area ; // area of the entire mesh
77    optimise.max_area = max_area ; // maximum area, i.e. area constraint
78
79    // Perform the optimisation
80    optimise.Solve_With_NewtonRaphson () ;
81
82    optimise.get_lambdas(lambdas);
83  }
```

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. 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. The velocities to grid points from boundary points are then extended or extrapolated by using the fast marching method. In practice, velocities are only extended to nodes in narrow band. Similarly, the gradients are computed. Hence, the level-set function is able to be updated.

```
1  // Extend boundary point velocities to all narrow band nodes
2  level_set.computeVelocities (boundary.points) ;
3
4  // Compute gradient of the signed distance function within the narrow band
5  level_set.computeGradients () ;
```

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 to solve the Eikonal equation.

```
// Update the level set function
bool is_reinitialised = level_set.update (time_step) ;

// Reinitialise the signed distance function, if necessary
if (!is_reinitialised) {
 // Reinitialise at least every 20 iterations
 if (n_reinit == 20) {
  level_set.reinitialise () ;
  n_reinit = 0 ;
 }

} else n_reinit = 0 ;

// Increment the number of steps since reinitialisation
n_reinit++ ;
```

A convergence check may terminate the algorithm before allowed maximum iterations are reached. The convergence check is not performed for the first five iterations of the algorithm. 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`.

```
// Converence criterion [Dunning_11_FINAL]:
// find the max relative distance over the past five iterations:
objective_values.push_back (sens.objective) ;
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)) ;
 }

}

   // Check if convergence has been met:
if ((relative_difference < max_diff) & (area < 1.001 * max_area)) break;
```

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 of objective function and constraint values are output, while the users are given the option to output each step's results during iterative calculations. The level set and area fraction results at each iteration can be written out into vtk files. Boundary

(a) Initial design (b) Final solution

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

segments can also be written to a text file.

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

// Print statistics to .txt file
history_file.open ("results/history/history.txt", ios_base::app) ;
history_file << n_iterations << "\t" << sens.objective << "\t" << area << "\n" ;
history_file.close () ;

// Write level set and area fractions to .vtk file
io.saveLevelSetVTK (n_iterations, level_set, false, false, "results/level_set") ;
io.saveAreaFractionsVTK (n_iterations, lsm_mesh, "results/area_fractions") ;

// Write level set, area fractions, and boundary segments to .txt file:
io.saveBoundarySegmentsTXT (n_iterations, boundary, "results/boundary_segments") ;
```

Results

Default initial design is shown in the Figure 3a, where the binary image is created based on level-set values ϕ. (black: $\phi \geq 0$ and white: $\phi < 0$). The structure converges to an optimal solution as given in Figure 3b after 224 iterations with a compliance value of 14.9 by using default parameters and convergence criterion given in the source file. The convergence history is illustrated in Figure 4.

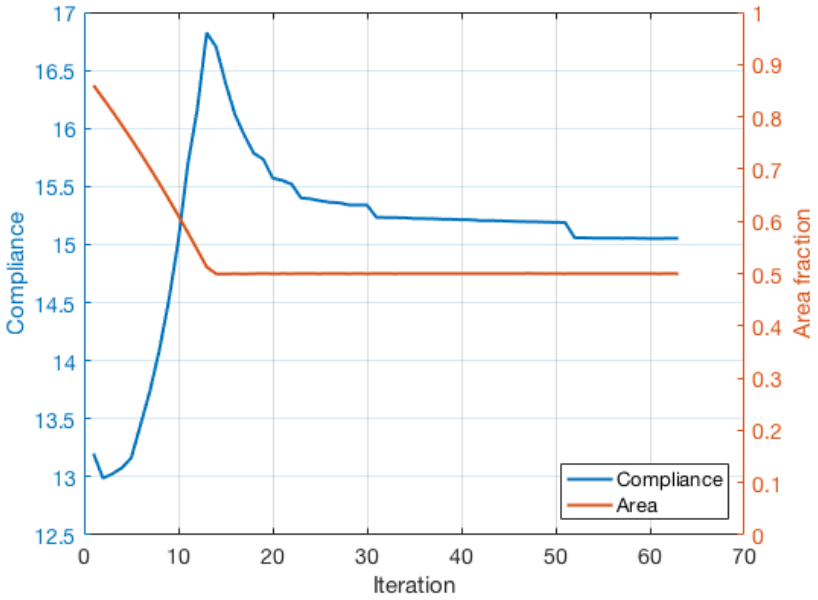


Figure 4: Convergence history of compliance value and area fraction for the cantilever example.

Tutorial 2: MBB Beam

This tutorial is considered an extension of **Tutorial 1**, so more considerations on the manipulation of boundary conditions and loads can be explored. Reader is recommended to first understand the previous tutorial before advancing through present example. Resources for this tutorial can be found in the folder **compliance** in /projects directory (same considered in **Tutorial 1**).

OpenLSTO code can be easily modified 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 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 as follows:

```
1     // Example 2: half of simply supported beam or MBB beam
2
3     // Left boundary condition
4   vector<double>        coord_left = {0.0, 0.0}, tol_left = {1e-12, 1e10} ;
5   vector<int>     fixed_nodes_left = fea_mesh.GetNodesByCoordinates (coord_left,
        tol_left) ;
6   vector<int> fixed_condition_left = {0} ; // set fixed in only the x direction.
7   vector<int>       fixed_dof_left = fea_mesh.dof (fixed_nodes_left,
        fixed_condition_left) ;
8
9   // Right boundary condition
10  vector<double>        coord_right = {nelx, 0.0}, tol_right = {1e-12, 1e-12} ;
11  vector<int>     fixed_nodes_right = fea_mesh.GetNodesByCoordinates(coord_right,
        tol_right) ;
12  vector<int> fixed_condition_right = {1} ; // set fixed in only the y direction.
13  vector<int>       fixed_dof_right = fea_mesh.dof(fixed_nodes_right,
        fixed_condition_right) ;
14
15  // Combine dofs into a single vector
16  vector<int> fixed_dof ;
17  fixed_dof.reserve(fixed_dof_left.size() + fixed_dof_right.size()) ;
18  fixed_dof.insert(fixed_dof.end(), fixed_dof_left.begin(), fixed_dof_left.end()) ;
19  fixed_dof.insert(fixed_dof.end(), fixed_dof_right.begin(), fixed_dof_right.end())
        ;
```

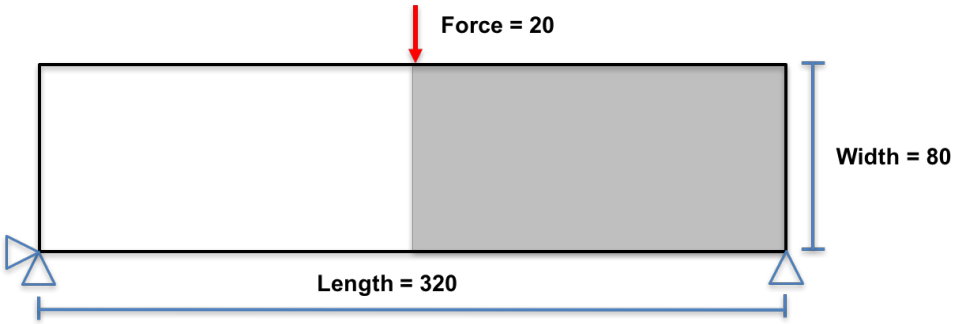


Figure 5: Configuration of the MBB beam.

(a) Initial design (b) Final solution

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

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.

```
// Example 2: half of simply supported beam or MBB beam

vector<double> coord = {0.0, nely}, tol = {1e-12, 1e-12} ;
vector<int>      load_node = fea_mesh.GetNodesByCoordinates (coord, tol) ;
vector<int> load_condition = {1} ; // apply load in only the y direction.
vector<int>       load_dof = fea_mesh.dof (load_node, load_condition) ;

vector<double> load_val (load_node.size()) ;
for (int i = 0 ; i < load_node.size() ; ++i) {
 load_val[i] = -10.0; //load component in y direction
}
```

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 86 iterations with a compliance value of 7497.9 by using default parameters and convergence criterion given in the source file. The convergence history is illustrated in Figure 7.

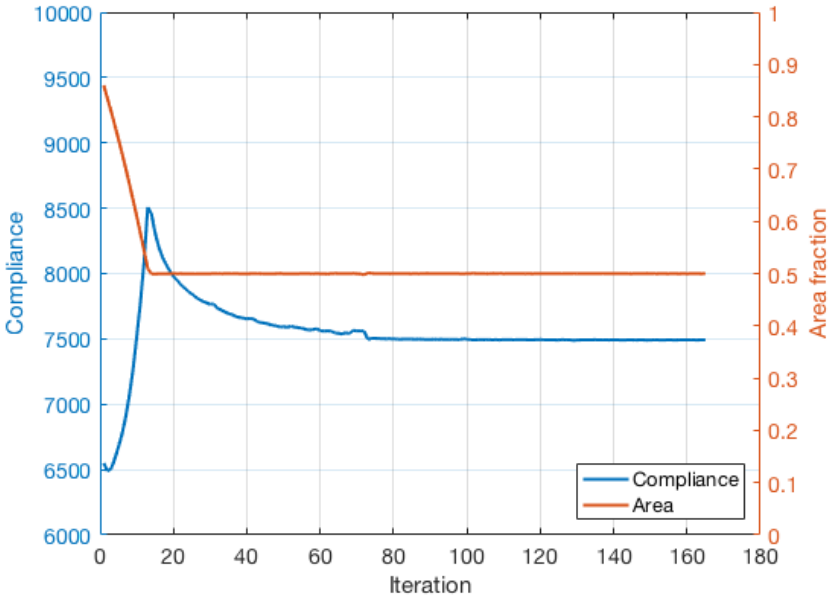


Figure 7: Convergence history of compliance value and area fraction for the MBB example.

Tutorial 3: 2D Stress Minimization

Goals

Upon completing this tutorial, the user will be familiar with performing a topology optimization for a stress minimization problem. The solution will provide an L-beam, which can be compared to solutions from other topology optimization approaches, such as solid isotropic material with penalization (SIMP) and bi-directional evolutionary structural optimization (BESO), as a validation case for OpenLSTO.

Resources

The resources for this tutorial can be found in the folder **stress_min** in OpenLSTO_v2/projects directory.

Tutorial

The following tutorial will walk you through the steps required to solve a stress minimization problem using OpenLSTO versus a compliance minimization problem. It is assumed that you have already obtained the OpenLSTO code and are familiar with the compliance minimization tutorial. If not, please refer to Download.

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 L-beam under a point load with configuration shown in Figure 8. It represents a modern benchmark to the assessing of stress minimization codes.

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;
5. the level set topology optimization loop.

Details for each part are explained below.

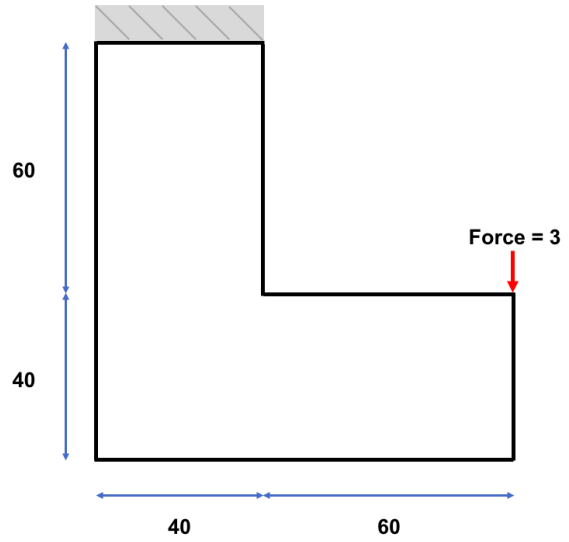


Figure 8: Configuration of the L-beam.

Settings for the Finite Element Analysis (Lines 22 to 128)

The number of elements in the x and y direction are each set to 100. The material properties are defined as follows:

```
1   double E = 1; // Young's modulus
2   double nu = 0.3; // Poisson's ratio
3   double rho = 1.0; // density
```

The Dirichlet boundary condition, i.e., clamped nodes, is defined at the top of the beam as follows:

```
1   coord = {0.0, num_elem_y};
2   tol = {num_elem_x + 0.1, 0.1};
3   vector<int> fixed_nodes = fea_mesh.GetNodesByCoordinates(coord, tol);
4   // Get degrees of freedom associated with the fixed nodes
5   vector<int> fixed_dof = fea_mesh.dof(fixed_nodes);
6   // Apply magnitude of displacement at the BC
7   vector<double> amplitude(fixed_dof.size(), 0.0); // Values equal to zero
```

The method `GetNodesByCoordinates`, in this case, selects all nodes that are within the distance `100.1` in the x-direction and `0.1` in the y-direction of the coordinate `(0.0, 100.0)`. `fea_mesh.dof(fixed_nodes)` selects the degrees of freedom associated with the selected nodes and `amplitude(fixed_dof.size(), 0.0)` sets the Dirichlet boundary condition as homogeneous.

A vertical point load is applied at the tip of the L-beam over two nodes as follows:

```
1   load_coord = {num_elem_x, double(num_elem_y)*2.0/5.0};
2   load_tol = {1.1, 0.1};
3   // Get vector of associated loads and degrees of freedom
4   vector<int> load_nodes = fea_mesh.GetNodesByCoordinates(load_coord, load_tol);
5   vector<int> load_dof = fea_mesh.dof(load_nodes);
6   // Assign magnitudes to the point load
7   uint num_load_nodes = load_nodes.size();
8   vector<double> load_values(load_nodes.size() * 2);
9   for (uint i = 1; i < num_load_nodes; i++) {
```

```
10      load_values[2*i] = 0.0; // x component
11      load_values[2*i+1] = -3.0 / double(num_load_nodes); // y component
12    }
13    FEA::PointValues point_load(load_dof, load_values);
```

In this case, the 2 nodes selected by `GetNodesByCoordinates` are within the distance `1.1` in the x-direction and `0.1` in the y-direction of coordinate `(100.0, 40.0)`. As such, the selected nodes are at coordinates `(99.0, 40.0)` and `(100.0, 40.0)`. The vector `load_values` contains the magnitudes of the loads applied at the selected nodes in the x and y directions. Therefore, the first two elements in the `load_values` vector will correspond to the magnitude of the load applied at the first node (first element in `load_nodes`) in the x and y direction respectively.

Settings for the Sensitivity Analysis (Lines 132 to 149)

Following lines are used to define sensitivity analysis parameters:

```
1   int sens_type = 1; // type of sensitivity being calculated. 0 is compliance, 1
2                      // is stress
3   double min_area_fraction = 0.1; // minimum element area to compute sensitvity
4   double least_sq_radius = 2.0;   // setting least square calculation to 2.0 grid
5                                   // spaces
6
7   // Initialize sensitvity analysis
8   FEA::SensitivityAnalysis sens(fea_study);
```

The variable `sens_type` defines the type of sensitivity being implemented, which in this case is stress. The variable `least_sq_radius` defines an important parameter for the least-squares interpolation scheme used in this level set implementation and the `FEA::SensitivityAnalysis` class is used to compute the sensitivity of stress with respect to the movements of the structural boundary.

Settings for the Level Set Analysis (Lines 153 to 244)

Following lines define the boundaries of the level set mesh for the L-shape geometry.

```
1   // Initialize the level set mesh
2   LSM::Mesh lsm_mesh(num_elem_x, num_elem_y, is_periodic);
3   // Define vectors of points for L-beam internal edges
4   vector<LSM::Coord> vertical_edge(2), horizontal_edge(2);
5   double inner_corner = double(num_elem_x)*2.0/5.0;
6   // Define a rectangle containing the points for the vertical edge
7   vertical_edge[0] = LSM::Coord({inner_corner - 0.01, inner_corner - 0.01});
8   vertical_edge[1] = LSM::Coord({inner_corner + 0.01, num_elem_y + 0.01});
9   // Define a rectangle containing the points for the horizontal edge
10  horizontal_edge[0] = LSM::Coord({inner_corner - 0.01, inner_corner - 0.01});
11  horizontal_edge[1] = LSM::Coord({num_elem_x + 0.01, inner_corner + 0.01});
12  // Define mesh boundary
13  lsm_mesh.createMeshBoundary(vertical_edge);
14  lsm_mesh.createMeshBoundary(horizontal_edge);
```

The method `createMeshBoundary` creates a boundary from the points within a rectangular region defined by two coordinates. In this case, `vertical_edge` is defined by coordinates `(39.99, 39.99)` and `(40.01, 100.01)` and `horizontal_edge` is defined by coordinates `(39.99, 39.99)` and `(100.01, 40.01)`.

The L-beam shape also needs to be defined for the level set itself. This is done as follows:

```
// Initialize the level set object
LSM::LevelSet level_set(lsm_mesh, holes, move_limit, band_width, is_fixed_domain);
// Kill level set nodes that aren't in L-beam region
vector<LSM::Coord> kill_region(2);
kill_region[0] = LSM::Coord({inner_corner + 0.01, inner_corner + 0.01});
kill_region[1] = LSM::Coord({num_elem_x + 0.01, num_elem_y + 0.01});
level_set.killNodes(kill_region);
// Define level set boundary (L-beam inner edges)
level_set.createLevelSetBoundary(vertical_edge);
level_set.createLevelSetBoundary(horizontal_edge);
```

The vector `kill_region` defines the square region containing the level set nodes that are outside of the L-beam. In this case, the region is defined by coordinates `(40.01, 40.01)` and `(100.01, 100.01)`. The method `killNodes` fixes the signed distance value of the nodes to a small negative value. To avoid issues that can arise from the singularity caused by point loads, it is beneficial to fix level set nodes surrounding, and including, nodes where the load has been applied. In this case, a total of 6 level set nodes are fixed using the following:

```
// Vector of points to fix Level set nodes (useful for load points)
double tol_x = 3.01, tol_y = 2.01;
double load_coord_x = num_elem_x, load_coord_y = double(num_elem_y)*2/5;
std::vector<LSM::Coord> points(2);
points[0] = LSM::Coord({load_coord_x - tol_x, load_coord_y - tol_y});
points[1] = LSM::Coord({load_coord_x + 0.01, load_coord_y + 0.01});
level_set.fixNodes(points);
points.clear();
```

Settings for the Optimization (Lines 248 to 273)

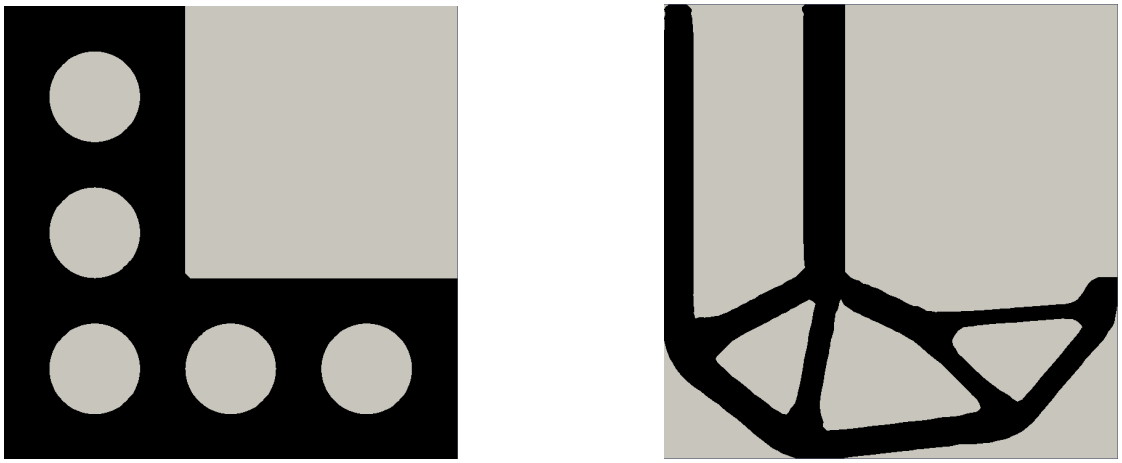
For the stress case, the following optimization parameters need to be adjusted or defined as follows:

```
double mesh_area = lsm_mesh.width * lsm_mesh.height - pow(double(lsm_mesh.width)*3/5, 2); // LSM mesh area
double p_norm = 6; // p_norm value for stress
```

The variable `p_norm` is the aggregation parameter p in the p-norm function $\left(\int_\Omega \sigma_{vm}^p d\Omega\right)^{\frac{1}{p}}$.

Level Set Topology Optimization Loop (Lines 277 to 470)

For the stress problem analysed in this tutorial, gauss point sensitivities need to be interpolated to the boundary points in the following way:



(a) Inital design (b) Final solution

Figure 9: Initial design and optimal solution for the L-beam example.

```
1   // Assign sensitivities to all the boundary points
2   for (int i = 0; i < boundary.points.size(); i++) {
3     // current boundary point
4     vector<double> boundary_point (2, 0.0);
5     boundary_point[0] = boundary.points[i].coord.x;
6     boundary_point[1] = boundary.points[i].coord.y;
7     // Interpolate Gauss point sensitivities by least squares
8     sens.ComputeBoundarySensitivities(boundary_point, least_sq_radius, sens_type,
          p_norm);
9     // Assign sensitvities
10    boundary.points[i].sensitivities[0] = -sens.boundary_sensitivities[i];
11    boundary.points[i].sensitivities[1] = -1;
12  }
```

Input `sens_type` of method `ComputeBoundarySensitivities` determines how (for the compliance or stress case) the sensitivities will be interpolated to the boundary points. Note that if `sens_type` is not defined the default interpolation case for `ComputeBoundarySensitivities` is compliance.

Results

For a given initial design as shown in Figure 9a, the structure converges to an optimal solution as given in Figure 9b after 177 iterations with a p-norm stress value of 3.53 and maximum von Mises stress value of 2.44 by using default parameters and convergence criteria given in the source file. The convergence history is illustrated in Figure 10.

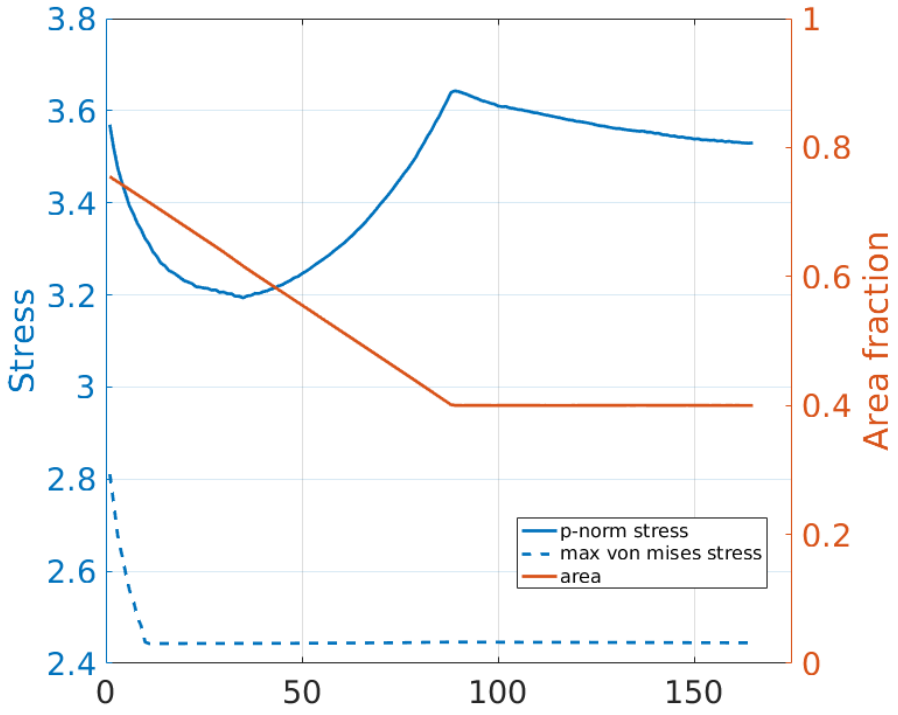


Figure 10: Convergence history of p-norm stress, maximum von Mises stress values and area fraction for the L-beam example.

Tutorial 4: Compliance Minimization Problem using Hole Creation Algorithm

Goals

In traditional level set method new holes are not allowed to be created naturally and optimized solutions are dependent on initial designs. Instead of calculating complex topological derivatives, a simple and efficient hole creation algorithm was introduced by our group [Dunning, P. D., Alicia Kim, H. (2013). A new hole insertion method for level set based structural topology optimization. International Journal for Numerical Methods in Engineering, 93(1), 118-134]. Inspired by the hole creation ability in 3D level set topology optimization problems, a secondary level set function representing a pseudo third dimension in two-dimensional problems to facilitate new hole insertion is introduced. The update of the secondary function is connected to the primary level set function forming a meaningful link between boundary optimization and hole creation.

For the implementations, some commands associated with the second level set function are added while most of commands about the primary level set function are not modified. In the following, only new added parts are introduced for concise.

Resources

The resources for this tutorial can be found in the folder **hole_creation** in OpenLSTO_v2/projects directory.

Tutorial

The following tutorial will walk you through the steps required to solve a compliance minimization problem with hole creation algorithm using the last OpenLSTO

update. It is assumed that you have already obtained this version of the code. If not, please refer to Download.

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 investigates a 2D cantilever beam under a point load as shown in Figure 11. This is exact the same problem presented in **Tutorial 1**, but additional explanations on the hole creation algorithm are given here.

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;

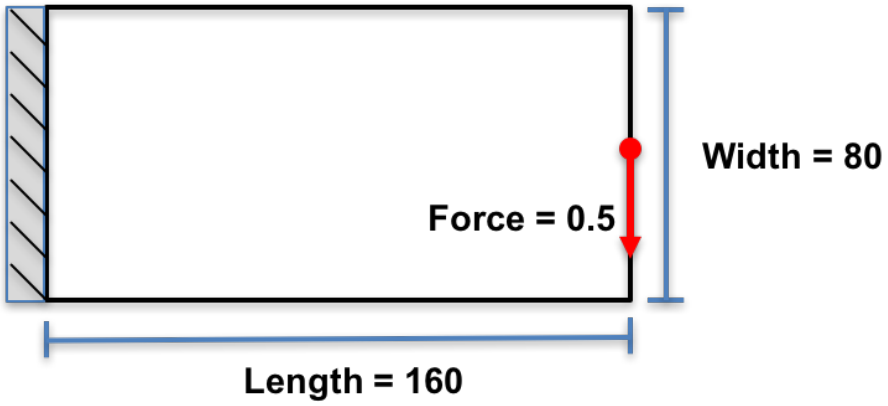


Figure 11: Configuration of the cantilever beam.

4. settings for the optimization;

5. the level set topology optimization loop.

Details for each part are explained below.

Settings for the Finite Element Analysis

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 12.

First, a finite element model is created through the following lines:

```
/*
 FEA Mesh:
*/

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

// Number of elements in x and y directions:
const unsigned int nelx = 160, nely = 80 ;

// fea_box contains the (x,y) coordinates of 4 corner points of rectangle
    containing the mesh:
MatrixXd fea_box(4,2);

fea_box <<   0.0,  0.0,
   nelx,  0.0,
   nelx, nely,
    0.0, nely;

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

  // Create structured mesh and assign degrees of freedom:
```

```
fea_mesh.MeshSolidHyperRectangle ({nelx, nely}, fea_box, element_order, false) ;
fea_mesh.is_structured = true ;
fea_mesh.AssignDof () ;
```

A two dimensional `FEA::Mesh` class, which will hold information pertaining to nodes, elements and degrees of freedom, is instantiated. A rectangular design domain is defined by its four corner points in the `fea_box` parameter, and the structured mesh is generated using the `MeshSolidHyperRectangle` function. The degrees of freedom are assigned using `AssignDof`.

Material properties such as Young's modulus, Poisson's ratio and density are added to the mesh as follows:

```
double   E = 1.0 ; // Young's Modulus
double  nu = 0.3 ; // Poisson's ratio
double rho = 1.0 ; // Density
```

An instance of the `FEA::StationaryStudy` class is created, which is capable of solving problems of the form $[K]\{u\} = \{f\}$. Dirichlet boundary conditions are defined in order to fix the degrees of freedom on the left-most edge. A point load is defined on the mid right-most edge. Since in this example the load vector $\{f\}$ is design-independent, the point load is built using the `AssembleF` function. These are realized in the following lines:

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

 FEA::StationaryStudy fea_study (fea_mesh) ;

/*
 Define homogeneous Dirichlet boundary condition (fixed nodes) and add to study:
*/

// Select dof using a box centered at coord of size tol:
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:
vector<double> amplitude (fixed_dof.size(),0.0) ; // Values equal to zero.
   fea_study.AddBoundaryConditions (FEA::DirichletBoundaryConditions (fixed_dof,
       amplitude, fea_mesh.n_dof)) ;

/*
 Define a point load of (0, -0.5) at the point (nelx, 0.5*nely) and add to study:
*/

 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) {
```

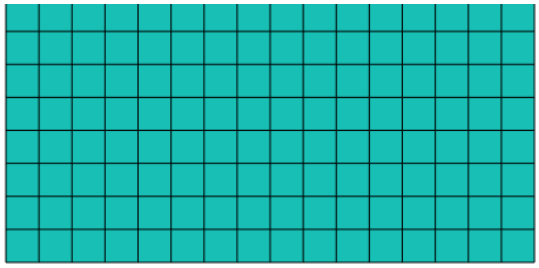


Figure 12: FEA and LSM meshing of design domain.

```
32      load_val[2*i]   = 0.00 ; // load component in x direction.
33      load_val[2*i+1] = -0.5 ; // load component in y direction.
34    }
35
36    // Add point load to study and assemble load vector {f}:
37    FEA::PointValues point_load (load_dof, load_val) ;
38    fea_study.AssembleF (point_load, false) ;
```

Lastly, the FEA solver settings (initial solution guess and convergence tolerance) are defined as:

```
1   // Initialise guess solution for CG:
2   vector<double> u_guess (fea_mesh.n_dof, 0.0) ;
3
4   // Convergence tolerance:
5   double cg_tolerence = 1.0e-6 ;
```

Settings for the Sensitivity Analysis

The `FEA::SensitivityAnalysis` 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 as:

```
1   FEA::SensitivityAnalysis sens (fea_study) ;
```

Settings for the Level Set Method

Basic parameters width related to level set method are defined first. They include `move_limit` defining that the level set boundary will move less than this value between design iterations, and `band_width` for the width of the narrow band: the area nearest the structural boundary which is re-initialized between iterations.

```
1   /*
2    Define LSM parameters:
3   */
4
5   double     move_limit = 0.5 ;   // Maximum displacement per iteration in units of
        the mesh spacing.
6   double     band_width = 6 ;     // Width of the narrow band.
7   bool is_fixed_domain = false ; // Whether or not the domain boundary is fixed.
```

It should be noted that the holes class is declared, however, holes initiated are inactive.

```
/*
 Seed initial holes:
 In this example, we create five horizontal rows, each row alternating between
 four and five equally spaced holes, all of radius 5 units.
*/

vector<LSM::Hole> holes ;

// First row with five holes:
// holes.push_back (LSM::Hole (16, 14, 5)) ;
// holes.push_back (LSM::Hole (48, 14, 5)) ;
// holes.push_back (LSM::Hole (80, 14, 5)) ;
// holes.push_back (LSM::Hole (112, 14, 5)) ;
// holes.push_back (LSM::Hole (144, 14, 5)) ;

// // Second row with four holes:
// holes.push_back (LSM::Hole (32, 27, 5)) ;
// holes.push_back (LSM::Hole (64, 27, 5)) ;
// holes.push_back (LSM::Hole (96, 27, 5)) ;
// holes.push_back (LSM::Hole (128, 27, 5)) ;

// // Third row with five holes:
// holes.push_back (LSM::Hole (16, 40, 5)) ;
// holes.push_back (LSM::Hole (48, 40, 5)) ;
// holes.push_back (LSM::Hole (80, 40, 5)) ;
// holes.push_back (LSM::Hole (112, 40, 5)) ;
// holes.push_back (LSM::Hole (144, 40, 5)) ;

// // Fourth row with four holes:
// holes.push_back (LSM::Hole (32, 53, 5)) ;
// holes.push_back (LSM::Hole (64, 53, 5)) ;
// holes.push_back (LSM::Hole (96, 53, 5)) ;
// holes.push_back (LSM::Hole (128, 53, 5)) ;

// // Fifth row with five holes:
// 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)) ;
```

Settings for the Optimization

Before solving the optimization problem, several parameters are set. `max_iterations` limits the number of design iterations; `max_area` defines the constraint of finite elements with area fractions, but it is ignored in the sensitivity analysis calculations; `max_diff` specifies the change in objective function required to signify convergence; `lambdas` are used to store weighting factors.

```
/*
 Define parameters needed for optimization loop:
*/

```

```
int    max_iterations = 300 ;    // maximum number of iterations.
double       max_area = 0.5 ;    // maximum material area.
double       max_diff = 0.0001 ; // relative difference between iterations must be
     less than this value to reach convergence.

/*
 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) ;
```

Level Set Topology Optimization Loop

An instance of an `LSM::Mesh` class is created 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 `LSM::Boundary` object.

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

double mesh_area = lsm_mesh.width * lsm_mesh.height ;

// 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 () ;

// Initialise the boundary object :
LSM::Boundary boundary (level_set) ;
    // Initialise random number generator:
LSM::MersenneTwister rng ;

/*
 Optimization:
*/

// Declare parameters that will change within the optimization loop:
unsigned int   n_reinit = 0 ;              // num cycles since signed dist
    reinitialisation.
double         time     = 0 ;              // running time.
vector<double> times, compliances, areas ; // time, compliance and area
    measurements.
int            n_iterations = 0 ;          // iteration counter
```

```
29  vector<double> objective_values ;          // vector to save objective history
30  double         relative_difference = 1.0 ; // convergence criteria variable,
31
32  // Initialise io object:
33  LSM::InputOutput io ;
```

Afterwards, the initial parameters for the hole creation algorithm are defined. `holeCFL` is the CFL condition and `lBand` is the band width for the second level set function. `h_bar` is the artificial height of the second level set function, which is further determined. `h_flag` is a variable implying whether hole should be inserted or not. `isHole` determines whether hole creation function is active nor not. `newHoleAreaLimit` is a threshold value for warning of moving too much material by the hole insertion process and initialised.

```
1   // ---------------------------------------------------------
2      // Define/Initialise parameters for hole insertion subroutine
3      vector <double> h_index(lsm_mesh.nNodes);
4      vector <double> h_lsf(lsm_mesh.nNodes);
5      vector <bool> h_elem(lsm_mesh.nElements);
6      vector <M2DO_LSM::h_Node> h_Nsens(lsm_mesh.nNodes);
7      vector <M2DO_LSM::h_Node> h_Nsens_Temp(lsm_mesh.nNodes);
8      vector <M2DO_LSM::h_Element> h_Esens(lsm_mesh.nElements);
9      int h_count = 0;
10     double holeCFL = 0.15;
11     double lBand = 2.0;
12     double h; // Size of level set element max (width, height)
13     double h_bar;  // artifitial height for setting secondary level set function
14     bool h_flag = false;  //
15     bool isHole = true; // Is hole inseration function active?
16     double newHoleAreaLimit = 0.03;
17     //
18     // Assign desired artificial height
19     //
20     h = ( lsm_mesh.width/nelx > lsm_mesh.height/nely ) ? (lsm_mesh.width/nelx) : (
           lsm_mesh.height/nely);
21     h_bar = h;
```

The boundary of the second level set function is to be defined as:

```
1      // LSM::Boundary_hole boundary_hole(levelSet) ;
2      LSM::Boundary boundary_hole(level_set) ;
```

In the next part of the program, the optimization loop that starts, and it is terminated with a convergence check when satisfactory solution is obtained. Iterations are counted by variable `n_iterations` and continue for a maximum of `max_iterations`. Inside the optimization loop, the boundary of the structure defined by iso-contour (2D) or surface (3D), e.g. zero level set, is discretized by finding intersection points on mesh grid with the use of marching square algorithm. The area fraction of each level set element is then calculated. The area fractions of elements are assigned to the finite elements. The area fraction fixed grid finite element method is called to conduct finite element analysis through assembling global stiffness matrix and solving finite element equation. The shape sensitivity of the compliance at each gauss points

in finite element is calculated. Shape sensitivities are then calculated for boundary points by extrapolating or interpolating from the information at gauss points with the use of the weighted least square method. The volume constraint is also imposed at each iteration. By completing these necessary inputs, the Lagrangian Multiplier method is applied to solve the optimization problem.

```
while (n_iterations < max_iterations) {

 ++n_iterations ;

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

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

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

  if (lsm_mesh.elements[i].area < 1e-3) {
   fea_mesh.solid_elements[i].area_fraction = 1e-3 ;
  }

  else {
   fea_mesh.solid_elements[i].area_fraction = lsm_mesh.elements[i].area ;
  }

 }

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

 // Solve equation using conjugant gradient (cg) method:
 fea_study.SolveWithCG();

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

 // Compute compliance sensitivities at boundary points:
 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 Gauss 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;

 }

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

 // Time step associated with the iteration
```

```
53    double time_step ;
54
55    // Constraint distance vector
56    vector<double> constraint_distances ;
57
58    // Push current distance from constraint violation into vector
59    constraint_distances.push_back (mesh_area * max_area - boundary.area) ;
60
61    /* Initialise the optimisation object
62
63    The Optimise class is a lightweight object so there is no cost for
64    reinitialising at every iteration. A smart compiler will optimise
65    this anyway, i.e. the same memory space will be reused. It is better
66    to place objects in the correct scope in order to aid readability
67    and to avoid unintended name clashes, etc.
68    */
69
70    LSM::Optimise optimise (boundary.points, time_step, move_limit) ;
71
72    // set up required parameters
73    optimise.length_x = lsm_mesh.width ;
74    optimise.length_y = lsm_mesh.height ;
75    optimise.boundary_area = boundary.area ; // area of structure
76    optimise.mesh_area = mesh_area ; // area of the entire mesh
77    optimise.max_area = max_area ; // maximum area, i.e. area constraint
78
79    // Perform the optimisation
80    optimise.Solve_With_NewtonRaphson () ;
81
82    optimise.get_lambdas(lambdas);
83  }
```

After optimizing the primary level set function in each step, the second level set function needs to be updated to determine whether and where new holes should be inserted. If so, the primary level set function should be modified with new holes. Such process can be achieved through following steps:

1. initialise the nodes values of the second level set function as an artificial height, `h_bar`, in terms of element length, `h`;

```
1   //
2   // Step 1.
3   //
4
5   h_count = hole_map(lsm_mesh, level_set, h, lBand, h_index, h_elem);
```

2. determine area/nodes where new holes can be inserted;

```
1   //
2   // Step 2. Initialise the secondary level set function after inserting new
        holes
3   //
4    h_lsf.resize(lsm_mesh.nNodes); fill(h_lsf.begin(), h_lsf.end(), h_bar);//}
5   get_h_lsf( lsm_mesh.nNodes, h_index, h_Nsens, lambdas, h_lsf );
6   h_flag = false;
```

3. calculate the corresponding nodal sensitivities needs to be calculated using least-squares method;

```
1
2  // 3.1 Extrapolate nodal sensitvities from sensitivites for gauss points
3  for (int inode = 0; inode < lsm_mesh.nNodes; inode++ ) {
4   vector<double> nPoint (2, 0);
5   nPoint[0] = lsm_mesh.nodes[inode].coord.x;
6   nPoint[1] = lsm_mesh.nodes[inode].coord.y;
7
8   // Interpolate Node Point sensitivities by least squares.
9      sens.ComputeBoundarySensitivities(nPoint) ;
10
11  // Assign sensitivities.
12  h_Nsens_Temp[inode].sensitivities.resize(2);
13  fill(h_Nsens_Temp[inode].sensitivities.begin(), h_Nsens_Temp[inode].
        sensitivities.end(), 0.0);
14
15  h_Nsens_Temp[inode].sensitivities[0] = -sens.boundary_sensitivities[inode]
        ;
16  h_Nsens_Temp[inode].sensitivities[1] = -1 ;
17 }
18 // clearing sens.boundarysens vector.
19 sens.boundary_sensitivities.clear() ;
20
21 // 3.2 Calculate element sensitivities
22 for (int iel = 0; iel < lsm_mesh.nElements; iel++ ) {
23   h_Esens[iel].sensitivities.resize(2);
24  fill(h_Esens[iel].sensitivities.begin(), h_Esens[iel].sensitivities.end(),
        0.0);
25
26     for (int ind = 0; ind < 4; ind++) {
27      int inode;
28      inode = lsm_mesh.elements[iel].nodes[ind];
29         h_Esens[iel].sensitivities[0] += 0.25 * h_Nsens_Temp[inode].
               sensitivities[0];
30         h_Esens[iel].sensitivities[1] += 0.25 * h_Nsens_Temp[inode].
               sensitivities[1];
31     }
32
33  }
34
35     // 3.3 Update nodal sensitivities
36  // clear nodal sensitivity value
37  for (int inode = 0; inode < lsm_mesh.nNodes; inode++ ) {
38  h_Nsens[inode].sensitivities.resize(2);
39  fill(h_Nsens[inode].sensitivities.begin(), h_Nsens[inode].sensitivities.end
        (), 0.0);
40  }
41
42 // re-assign nodal sensitivity value based on calculatd element
       sensitivities
43 for (int iel = 0; iel < lsm_mesh.nElements; iel++ ) {
44  for (int ind = 0; ind < 4; ind++) {
45   int inode;
46   inode = lsm_mesh.elements[iel].nodes[ind];
47   h_Nsens[inode].sensitivities[0] += 0.25 * h_Esens[iel].sensitivities[0];
48   h_Nsens[inode].sensitivities[1] += 0.25 * h_Esens[iel].sensitivities[1];
49  }
```

```
50  }
```

4. update the secondary level set function for hole insertable nodes to check whether new hole should be inserted or not;

```
1       get_h_lsf( lsm_mesh.nNodes, h_index, h_Nsens, lambdas, h_lsf );
2       //
3       // Step 5. Check whether new holes should be inserted
4       //
5       int inserted_hole_nodes = 0;
6       for (int inode = 0; inode < lsm_mesh.nNodes; inode++ ) {
7           if ( (h_index[inode] ==1) && (h_lsf[inode] < 0) ) {
8               if (!h_flag) {
9                   cout << "\n\n-----------------------------------------\n";
10                  cout << "  Hole will be inserted at ";
11              }
12              h_flag = true;
13              inserted_hole_nodes ++;
14              //cout<< "\n" << inode << "\t" << h_lsf[inode] << "\t" <<
                    level_set.signedDistance[inode];
15           }
16      }
17      // counting hole_area secondary
18      double area_h_lsf, area_lsf;
19      area_h_lsf = LSM::Boundary_hole(level_set, &h_lsf).computeAreaFractions()
            ;
20      area_lsf   = LSM::Boundary_hole(level_set, &level_set.signedDistance).
            computeAreaFractions();
21
22      printf("\nThe area fraction corresponding to lsf and h_lsf are: %8.2f
            %8.2f %8.2f\n", boundary.area, area_lsf, area_h_lsf) ;
23
24      //cout << "\n\nNumber of nodes to be set as new hole nodes: " <<
            inserted_hole_nodes << endl;
```

5. copy the secondary level set function to the primary level set function if holes need to be inserted;

```
1       //
2       // Step 6. Copy values of secondary level set function to primaray level
            set function
3       //
4       // io.saveLevelSetVTK(9000, level_set) ;
5
6       if (h_flag) {
7
8           //
9           // 6.1 Check whether new holes' area exceeds a certain threshold
10          //
11
12          double hole_areafraction = (mesh_area- area_h_lsf)/area_lsf;
13          // Find minimum of h_lsf
14          double temp_min_h_lsf = 1.0;
15          if (hole_areafraction > newHoleAreaLimit) {
16              for (int inode = 0; inode < lsm_mesh.nNodes; inode++) {
17                  temp_min_h_lsf = (temp_min_h_lsf < h_lsf[inode]) ?
                        temp_min_h_lsf : h_lsf[inode];
18              }
```

```
19          }
20          while (hole_areafraction > newHoleAreaLimit) {
21
22              cout << "\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n";
23              cout << "     Too much material is removed. \n";
24              cout << "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!" << endl;
25
26              // move up h_lsf untill the limit of the area of new holes to be
                    inserted meeting the requirement
27              for (int inode = 0; inode < lsm_mesh.nNodes; inode++) {
28                  h_lsf[inode] = h_lsf[inode] - 0.005*temp_min_h_lsf;
29              }
30              area_h_lsf = LSM::Boundary_hole(level_set, &h_lsf).
                    computeAreaFractions();
31              hole_areafraction = (mesh_area- area_h_lsf)/area_lsf;
32          }
33
34          //
35          // 6.2 Update primary level set function to insert new holes
36          //
37          signedDistance_temp.clear(); double min_h_lsf = 1.0, min_lsf = 1.0;
38          for (int inode = 0; inode < lsm_mesh.nNodes; inode++) {
39              // signedDistance_temp[inode] = level_set.signedDistance[inode];
40              if ( (h_index[inode] ==1) && (hole_areafraction>1e-3)) {
41                  //if (h_lsf[inode] < level_set.signedDistance[inode]) {
42                   if ((h_lsf[inode]<=h_bar) && (h_lsf[inode] < level_set.
                         signedDistance[inode])) {
43                      level_set.signedDistance[inode] = h_lsf[inode];
44                  }
45              }
46              signedDistance_temp[inode] = level_set.signedDistance[inode];
47              min_h_lsf = (min_h_lsf < h_lsf[inode]) ? min_h_lsf : h_lsf[inode
                    ];
48              min_lsf = (min_lsf < level_set.signedDistance[inode]) ? min_lsf :
                     level_set.signedDistance[inode];
49          }
50          cout <<"\n\nMininal primary and secondary LSF: " << min_h_lsf << "\t"
                 << min_lsf << endl;
51      }
```

6. stretch this updated primary level set function using Fast Marching Method after inserting new holes.

```
1       //
2       // Step 7. Use fast marching method to re-initialise signed distance
            function
3       //
4       if (h_flag) {
5
6           // for (int inode = 0; inode < lsm_mesh.nNodes; inode++ ) {
7           //     level_set.signedDistance[inode] = h_lsf[inode];
8           // }
9           // cout << "\nThe area fraction corresponding to h_lsf is [new] : "
                << LSM::Boundary_hole(level_set,&h_lsf).computeAreaFractions() <<
                endl;
10          // io.savelevel_setVTK(9001, level_set) ;
11
12          // for (int inode = 0; inode < lsm_mesh.nNodes; inode++ ) {
13          //     level_set.signedDistance[inode] = signedDistance_temp[inode];
```

```
14          // }
15          // io.savelevel_setVTK(9002, level_set) ;
16
17          // // boundary_hole2.computeAreaFractions() ;
18          // cout << "\nThe area fraction corresponding to lsf before
                stretching is: " << LSM::Boundary_hole(level_set,&level_set.
                signedDistance).computeAreaFractions() << endl;
19
20          M2DO_LSM::FastMarchingMethod fmm(lsm_mesh, false);
21          fmm.march(level_set.signedDistance);
22
23          // io.savelevel_setVTK(9003, level_set) ;
24
25          // cout << "\nhe area fraction corresponding to lsf after stretching
                is: " << LSM::Boundary_hole(level_set,&level_set.signedDistance).
                computeAreaFractions() << endl;
26
27      }
```

In order to update the primary level set function, velocity values are required at all grid nodes. In practice, velocities are only extended to nodes in narrow band. Similarly, gradients are computed. Hence, the level set function is able to be updated.

```
1  // Extend boundary point velocities to all narrow band nodes
2  level_set.computeVelocities (boundary.points, time_step, 0, rng) ;
3
4  // Compute gradient of the signed distance function within the narrow band
5  level_set.computeGradients () ;
6
7  // Update the level set function
8  bool is_reinitialised = level_set.update (time_step) ;
```

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 to solve the Eikonal equation.

```
1  // Update the level set function
2  bool is_reinitialised = level_set.update (time_step) ;
3
4  // Reinitialise the signed distance function, if necessary
5  if (!is_reinitialised) {
6   // Reinitialise at least every 20 iterations
7   if (n_reinit == 20) {
8    level_set.reinitialise () ;
9    n_reinit = 0 ;
10  }
11
12 } else n_reinit = 0 ;
13
14 // Increment the number of steps since reinitialisation
```

```
15  n_reinit++ ;
```

A convergence check may terminate the algorithm before allowed maximum iterations are reached. The convergence check is not performed for the first five iterations of the algorithm. 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`.

```
1  // Converence criterion [Dunning_11_FINAL]:
2  // find the max relative distance over the past five iterations:
3  objective_values.push_back (sens.objective) ;
4  double objective_value_k, objective_value_m ;
5
6  if (n_iterations > 5) {
7
8   objective_value_k = sens.objective ;
9   relative_difference = 0.0 ;
10
11   for (int i = 1 ; i <= 5 ; i++) {
12    objective_value_m = objective_values[n_iterations - i - 1] ;
13    relative_difference = max(relative_difference, abs((objective_value_k -
          objective_value_m)/objective_value_k)) ;
14   }
15
16  }
17
18     // Check if convergence has been met:
19  if ((relative_difference < max_diff) & (area < 1.001 * max_area)) break;
```

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 of objective function and constraint values are output, while the users are given the option to output each step's results during iterative calculations. The level set and area fraction results at each iteration can be written out into vtk files. Boundary segments can also be written to a text file.

```
1  // Print statistics
2  printf ("%8.1f %12.4f %10.4f\n", double(n_iterations), sens.objective, area) ;
3
4  // Print statistics to .txt file
5  history_file.open ("results/history/history.txt", ios_base::app) ;
6  history_file << n_iterations << "\t" << sens.objective << "\t" << area << "\n" ;
7  history_file.close () ;
8
9  // Write level set and area fractions to .vtk file
10 io.saveLevelSetVTK (n_iterations, level_set, false, false, "results/level_set") ;
11 io.saveAreaFractionsVTK (n_iterations, lsm_mesh, "results/area_fractions") ;
12
13 // Write level set, area fractions, and boundary segments to .txt file:
```

(a) Initial design (b) Final solution

Figure 13: Initial design and optimal solution for the cantilever example.

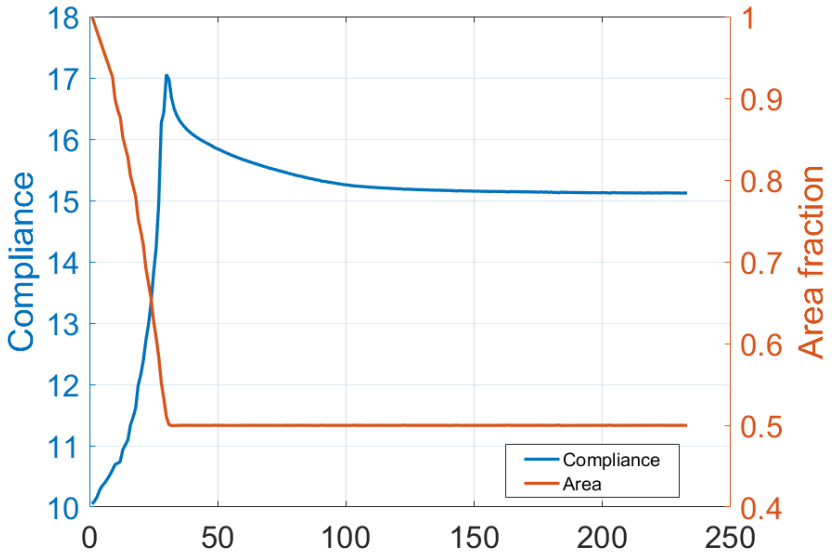


Figure 14: Convergence history of compliance value and area fraction for the cantilever example.

```
14    io.saveBoundarySegmentsTXT (n_iterations, boundary, "results/boundary_segments") ;
```

Results

For a given initial design as shown in Figure 13a, the structure converges to an optimal solution as given in Figure 13b after 233 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 14.

Tutorial 5: MBB Beam using the Hole Creation Algorithm

This tutorial is considered an extension of **Tutorial 4**, so more considerations on the manipulation of boundary conditions and loads can be explored. Reader is recommended to first understand the previous tutorial before advancing through present example. The resources for this tutorial can be found in the folder **hole_creation** in OpenLSTO_v2/projects directory (same considered in **Tutorial 4**).

The code can be easily modified 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 15 is demonstrated here. Only 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 as follows:

```
1     // Example 2: half of simply supported beam or MBB beam
2
3     // Left boundary condition
4   vector<double>        coord_left = {0.0, 0.0}, tol_left = {1e-12, 1e10} ;
5   vector<int>     fixed_nodes_left = fea_mesh.GetNodesByCoordinates (coord_left,
        tol_left) ;
6   vector<int> fixed_condition_left = {0} ; // set fixed in only the x direction.
7   vector<int>       fixed_dof_left = fea_mesh.dof (fixed_nodes_left,
        fixed_condition_left) ;
8
9   // Right boundary condition
10  vector<double>        coord_right = {nelx, 0.0}, tol_right = {1e-12, 1e-12} ;
11  vector<int>     fixed_nodes_right = fea_mesh.GetNodesByCoordinates(coord_right,
        tol_right) ;
12  vector<int> fixed_condition_right = {1} ; // set fixed in only the y direction.
13  vector<int>       fixed_dof_right = fea_mesh.dof(fixed_nodes_right,
        fixed_condition_right) ;
14
15  // Combine dofs into a single vector
16  vector<int> fixed_dof ;
17  fixed_dof.reserve(fixed_dof_left.size() + fixed_dof_right.size()) ;
```

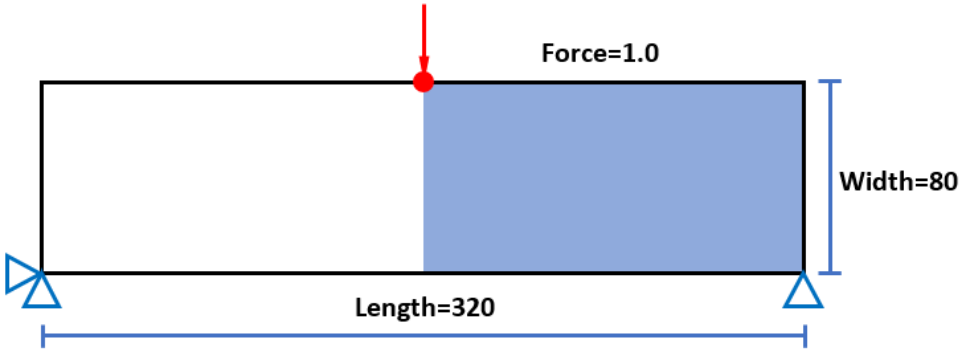


Figure 15: Configuration of the MBB beam.

(a) Initial design (b) Final solution

Figure 16: Initial design and optimal solution for the MBB example.

```
18  fixed_dof.insert(fixed_dof.end(), fixed_dof_left.begin(), fixed_dof_left.end()) ;
19  fixed_dof.insert(fixed_dof.end(), fixed_dof_right.begin(), fixed_dof_right.end())
        ;
```

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.

```
1   // Example 2: half of simply supported beam or MBB beam
2
3   vector<double> coord = {0.0, nely}, tol = {1e-12, 1e-12} ;
4   vector<int>      load_node = fea_mesh.GetNodesByCoordinates (coord, tol) ;
5   vector<int> load_condition = {1} ; // apply load in only the y direction.
6   vector<int>       load_dof = fea_mesh.dof (load_node, load_condition) ;
7
8   vector<double> load_val (load_node.size()) ;
9   for (int i = 0 ; i < load_node.size() ; ++i) {
10   load_val[i] = -10.0; //load component in y direction
11  }
```

After these setting, the optimization for MBB can be solved. For the given initial design as shown in Figure 16a, the structure converges to an optimal solution as given in Figure 16b after 300 iterations with a compliance value of 18.9 by using default parameters and convergence criterion given in the source file. The convergence history is illustrated in Figure 17.

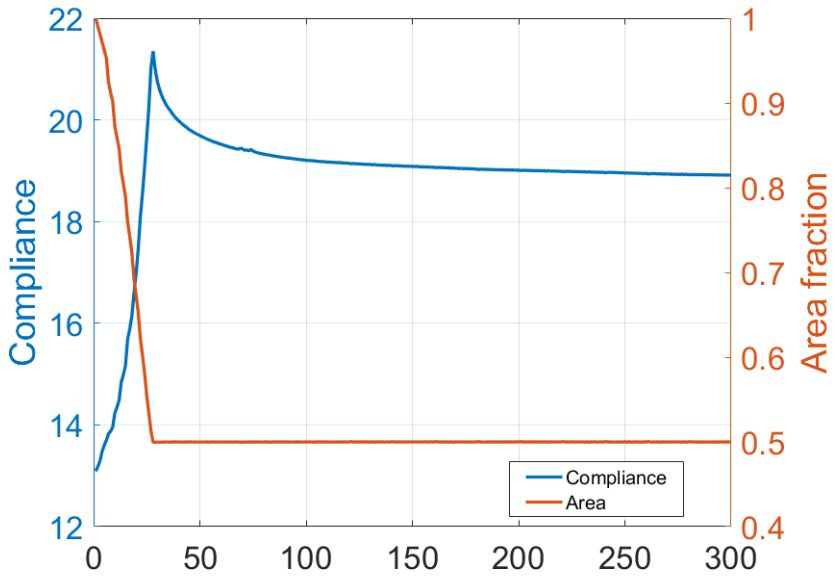


Figure 17: Convergence history of compliance value and area fraction for the MBB example.

Tutorial 6: 3D Level Set Topology Optimization

Goals

Upon completing this tutorial, the user will be familiar with performing a topology optimization for a 3D compliance minimization problem. The outline and functions used in this tutorial are very similar to the 2D compliance minimization and stress minimization problems, previously presented in tutorials 1 to 3.

Resources

The resources for this tutorial are found in the folder **3d** in OpenLSTO_v2/projects directory. The compliance minimization example needs to be compiled using `make comp_min` respectively; and must be executed using the command `./bin/a.out`.

Tutorial

The following tutorial will walk you through the steps required to solve a compliance minimization problem (`comp_min.cpp`) using OpenLSTO in 3D, subject to a volume constraint of 30%. It is assumed that you have already obtained the OpenLSTO code. If not, please refer to Download. It is also assumed that you are familiar with the compliance minimization code, introduced in Tutorial 1: Compliance Minimization Problem for 2D.

A schematic of the problem is shown in Figure 18. The different blocks of code that solve this optimization problem are introduced below.

Setting Up the Finite Element Analysis

The following code snippet is used to create a mesh of size $40 \times 20 \times 20$:

```
1
2 // Dimensionality of problem:
```

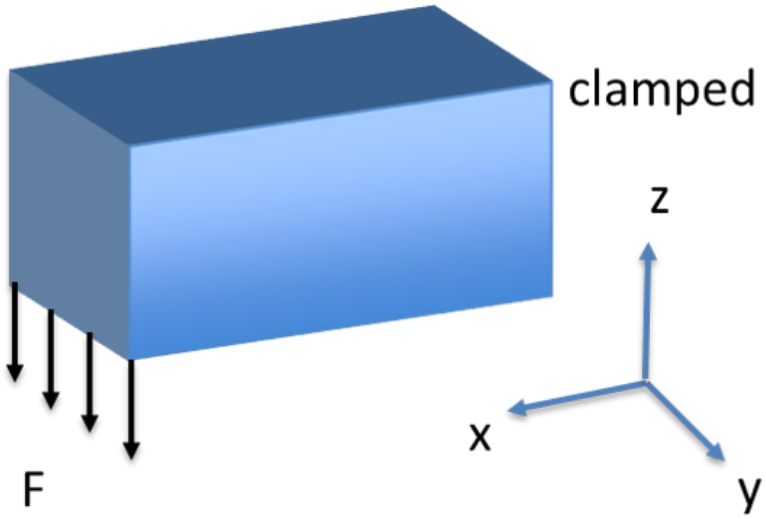


Figure 18: A schematic of the 3D compliance minimization problem. The structure is clamped on the right side and a distributed force is applied on the bottom-left side.

```
3  const int spacedim = 3 ;
4
5  // FEA & level set mesh parameters:
6  const unsigned int nelx = 40, nely = 20, nelz = 20;
7
8  // Create an FEA mesh object.
9  FEA::Mesh fea_mesh (spacedim) ;
10
11 // Mesh a hyper rectangle.
12 MatrixXd fea_box (8, 3) ;
13
14 fea_box <<   0.0, 0.0, 0.0,
15            nelx, 0.0 , 0.0,
16            nelx, nely , 0.0,
17            0.0, nely ,0.0 ,
18            0.0, 0.0 ,nelz,
19            nelx, 0.0 ,nelz,
20            nelx, nely ,nelz,
21            0.0, nely ,nelz;
```

It is clear that, as an extension of the 2D mesh creation code introduced in Tutorial 1: Compliance Minimization Problem for 2D, `spacedim` refers to the number of dimensions of a 3D geometry and `fea_box` receives 8 coordinates corresponding to the 8 vertices of the cuboid schematized in Figure 18.

After creating the mesh, degrees of freedom are assigned to each node as follows:

```
1 vector<int> nel = {nelx, nely , nelz} ;
2 int element_order = 3 ;
3 fea_mesh.MeshSolidHyperRectangle (nel, fea_box, element_order, false) ;
4 fea_mesh.is_structured = true ;
5 fea_mesh.AssignDof () ;
```

The geometry subject to the level set optimization has the following properties:

- Young's modulus: 1.0;
- Poisson's ratio: 0.3;
- density: 1.0.

and the code that adds these material properties to the analysis is written as follows:

```
// Add material properties:
// SolidMaterial (<geometry dimension of structure>, <Young's modulus>, <Poisson's ratio>, <density>)
fea_mesh.solid_materials.push_back (FEA::SolidMaterial (spacedim, 1.0, 0.3, 1.0)) ;
```

Next, we will define a study (analysis) and apply boundary conditions (clamped nodes located on plane $x = 0$):

```
// Define study
FEA::StationaryStudy fea_study (fea_mesh) ;
// Fix nodes
vector<int> fixed_nodes = fea_mesh.GetNodesByCoordinates ({0.0, 0.0 , 0.0 }, {1e-12, 1e9, 1e9}) ;
// Fix DOFs
vector<int> fixed_dof = fea_mesh.dof (fixed_nodes) ;
// Define Amplitude
vector<double> amplitude (fixed_dof.size(),0.0) ; // Values equal to zero.
// Add boundary conditions
fea_study.AddBoundaryConditions (FEA::DirichletBoundaryConditions (fixed_dof, amplitude, fea_mesh.n_dof)) ;
```

A distributed load, as described in Figure 18, is applied as follows:

```
// Select Nodes
vector<int> load_node = fea_mesh.GetNodesByCoordinates ({nelx, 0.0*nely,0.0*nelz}, {1.0e-12, 1.0e9, 1.e-12}) ;
    vector<int> load_dof  = fea_mesh.dof (load_node) ;

// Select DOFs
vector<double> load_val (load_node.size() * spacedim) ;

// Apply loads on nodes
for (int i = 0 ; i < load_node.size() ; ++i) {

        load_val[spacedim*i]   = 0.0 ;
        load_val[spacedim*i+1] = 0.0 ;
    load_val[spacedim*i+2] = -1.0 ;
}

// Assemble load vector
FEA::PointValues point_load (load_dof, load_val) ;
    fea_study.AssembleF (point_load, false) ;

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

Setting Up the Level Set Analysis

The following bits of code are used to create a level set mesh (which is of the same size as the finite element mesh) and a boundary of a box:

```
//  Create an object of the levelset class
LevelSet3D level_set_3d;

// Declare box dimensions and initialize the box
std::vector<double> LS2FEmap(3,1);

uint box_x = nelx*LS2FEmap[0];
uint box_y = nely*LS2FEmap[1];
uint box_z = nelz*LS2FEmap[2];

level_set_3d.SetBoxDimensions(box_x,box_y,box_z);// Set up dimensions

// Gotta define the pointers to phi and grid_vel in the main, otherwise it gets
    deleted!
level_set_3d.phi = new mp4Vector[level_set_3d.num_grid_pts];

// level_set_3d.holes.push_back({ 0.0*box_x, 0.5*box_y, 0.5*box_z, 0.4*box_y });
level_set_3d.MakeBox() ;
```

Handling Sensitivity Data

The class `SensitivityData` is used to transfer the sensitivity and optimization informations, such as volume constraint, move limit, optimum velocities and etc., between finite element analysis and level set classes. It is set up using the following code:

```
// Create a sensitivity object
SensitivityData SensData;

double MaxVol = 30.0; // in percentage
SensData.MaxVol = MaxVol;

std::vector<double> UB(2,0);
std::vector<double> LB(2,0);

// pass mesh dimensions to sensitivity data
SensData.nx = box_x;
SensData.ny = box_y;
SensData.nz = box_z;

SensData.LS2FEmap = LS2FEmap;

SensData.LB = LB;
SensData.UB = UB;

double move_limit = 0.25;
SensData.move_limit = move_limit;
```

The optimization iterations update the level set until the maximum allowed iterations or the convergence criteria is satisfied. The code shown in the following paragraphs is used to do the optimization:

After the discretization of the level set, boundary points and the volume fractions are computed from the level set using the following code:

```
// Discretize boundary using Marching Cubes
```

```
if(n_iterations > 1) delete[] level_set_3d.triangle_array;
level_set_3d.MarchingCubesWrapper();

// Pass info to SensData
SensData.bpointsize = level_set_3d.num_boundary_pts;
SensData.bPoints = level_set_3d.boundary_pts_one_vector;
SensData.pointAreas = level_set_3d.boundary_areas;
SensData.bpointsize = level_set_3d.num_triangles;
SensData.iter = n_iterations;

// Set-up narrow band
level_set_3d.SetupNarrowBand();


// Calculate volume fractions
level_set_3d.CalculateVolumeFractions();

SensData.volumeFractions = level_set_3d.volumefraction_vector;
```

The finite element analysis routine uses the volume fraction information to calculate the deflections and sensitivities as follows:

```
// Assign area fractions.
for (unsigned int i=0 ; i< fea_mesh.solid_elements.size() ; i++)
{
if (SensData.volumeFractions[i] < 1e-3) fea_mesh.solid_elements[i].area_fraction =
    1e-3 ;
else fea_mesh.solid_elements[i].area_fraction = SensData.volumeFractions[i] ;
}

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

// Solve equation:
fea_study.SolveWithCG () ;

SensData.compliance = fea_study.u.dot(fea_study.f);

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

After calculating the sensitivities, the least-squares interpolation scheme can be used to compute sensitivity at the boundary points and these sensitivities are passed on to the sensitivity object `SensData` as described below:

```
for (int i=0 ; i < SensData.bpointsize ; i++)
{
// current boundary point
std::vector<double> boundary_point(3);
boundary_point[0] = SensData.bPoints[3*i];
boundary_point[1] = SensData.bPoints[3*i+1];
boundary_point[2] = SensData.bPoints[3*i+2];

// compute boundary sensitivity at this point
sens.ComputeBoundarySensitivities(boundary_point) ;

// Assign sensitivities.
SensData.bsens[i] = -sens.boundary_sensitivities[i] ;
SensData.vsens[i] = -1 ;
```

```
15
16 // assign large values to sensitivities along the boundary where load is applied
17 if(boundary_point[0] >= nelx - 2 && boundary_point[2] <= 2 ) SensData.bsens[i] =
       1.0e5 ;
18 }
```

The sensitivity information is used to optimize the boundary velocities. The optimum velocities are then assigned to the `level_set_3d` object.

```
1 // Optimize boundary point movement
2 PerformOptimization(SensData);
3
4 // Resize and assign optimum velocities
5 level_set_3d.opt_vel.resize(level_set_3d.num_boundary_pts);
6
7 level_set_3d.opt_vel = SensData.opt_vel_nlopt;
```

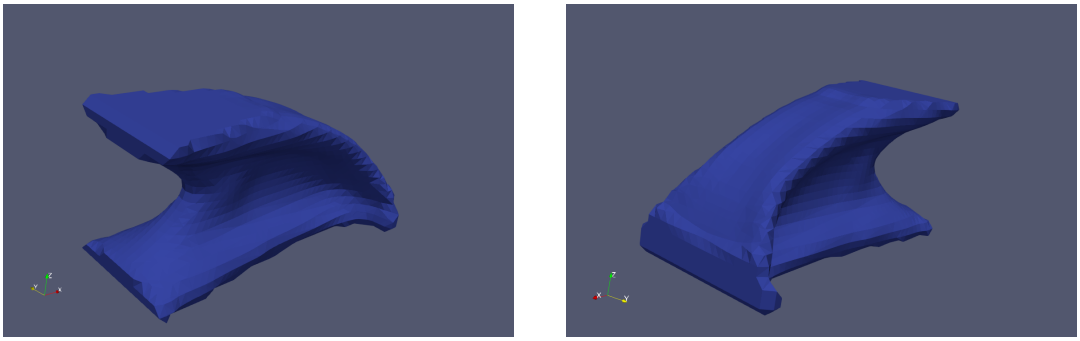
The level set is updated using the optimized velocities. First, the boundary velocities are extrapolated to the grid points. Next the fast marching method (FMM) is used to update the level set. These tasks are coded as follows:

```
1 // Extrapolate velocities
2 level_set_3d.ExtrapolateVelocities();
3
4 // Extend velocities using fast marching method
5 // FMM inside...
6 level_set_3d.indices_considered = level_set_3d.indices_considered_inside;
7 level_set_3d.FastMarchingMethod();
8
9 // FMM outside...
10 for(int i = 0; i < level_set_3d.num_grid_pts; i++)   level_set_3d.phi_temp[i] = -
       level_set_3d.phi_temp[i]; // flip sign
11 level_set_3d.indices_considered = level_set_3d.indices_considered_outside;
12 level_set_3d.FastMarchingMethod();
13 for(int i = 0; i < level_set_3d.num_grid_pts; i++) level_set_3d.phi_temp[i] = -
       level_set_3d.phi_temp[i]; // flip sign
14
15 // Advect
16 level_set_3d.Advect();
```

Output

The output of the code is a stl file, which can be visualized using ParaView (Figure 19). The following code creates the stl file:

```
1 // write to stl
2 int box_smooth = 1;
3 level_set_3d.WriteSTL(box_smooth);
```



(a) (b)

Figure 19: Optimized design for the cantilever example.