LS-TaSC™

TOPOLOGY AND SHAPE COMPUTATIONS FOR LS-DYNA®

USER'S MANUAL

April 2011
Version 2.0

Corporate Address
Livermore Software Technology Corporation
P. O. Box 712
Livermore, California 94551-0712

Support Addresses

Livermore Software Technology Corporation
7374 Las Positas Road
Livermore, California 94551
Tel: 925-449-2500 ♦ Fax: 925-449-2507
Email: sales@lstc.com
Website: www.lstc.com

Livermore Software Technology Corporation
1740 West Big Beaver Road
Suite 100
Troy, Michigan 48084
Tel: 248-649-4728 ♦ Fax: 248-649-6328

Disclaimer

14-Apr-11

PREFACE TO VERSION 2

Version 2 was started in spring of 2010 in response to industrial feedback regarding version 1. Version 2 is an important step forward containing the following major new features:

- Shell structure support
- Global constraints
- Multiple parts
- Symmetry definitions
- Casting direction definitions

Some minor features are:

- Tetrahedral solid element and triangular shell element support
- The speed of some algorithms was improved
- Improved integration with LS-DYNA

Many thanks are due to David Björkevik, who did the GUI design and implementation, as well as Tushar Goel, who did the initial global constraints implementation. Valuable feedback from customers and co-workers is also acknowledged.

Willem Roux
Livermore CA,
January 2011

PREFACE TO VERSION 1

The development of the topology code started in the fall of 2007 in response to a request from a vehicle company research group. The alpha version was released in the spring of 2009 to allow the vehicle company research groups to give feedback from an industrial perspective, while the beta version was released in November 2009.

Most of the methodology developments in version 1.0 are due to Tushar Goel who worked on the engine implementation and algorithm design. Additionally, he also wrote the manual together with Willem Roux.

The project architecture was the responsibilities of Willem Roux and David Björkevik. David had the lead role with regard to the graphical user interface aspects, while Willem had the senior role looking after the overall project and the project management.

Thanks are also due to Nielen Stander from LSTC who helped to coordinate the efforts in the LS-OPT group and sourced the initial version of the technology, John Renaud and Neal Patel for discussion regarding topology optimization, Kishore Pydimarry and Ofir Shor for evaluating the alpha version, and Fabio Mantovani and Stefano Mazzalai for their help with LS-DYNA simulations.

Willem Roux
Livermore CA,
January 2010

TABLE OF CONTENTS

Preface to Version 2 3
Preface to Version 1 4
Table of Contents 5
Table of Figures 9
1. Introduction 10
1.1. Classification of Structural Optimization Techniques 10
1.1.1. Topology Optimization 10
1.1.2. Topometry Optimization 10
1.1.3. Size Optimization 10
1.1.4. Shape Optimization 10
1.2. Brief Overview 10
1.3. Topology Optimization Method in LS-TaSC 11
1.4. References 12
2. User's Manual 13
2.1. Running the Program 13
2.2. Design Goal 13
2.3. Problem Definition 13
2.4. The Design Parts 13
2.4.1. Elementwise Material Density and Element Deletion for Solids 14
2.4.2. Design of Shells 14
2.4.3. Element types 14
2.4.4. Material data 14
2.5. Geometry and manufacturing definitions 15
2.6. Design Variables 16
2.6.1. Mapping Elements to the Design Variables 16
2.6.2. Filtering of Results 16
2.6.3. Initialization of the Design Variables 17
2.7. LS-DYNA® Specifics 17
2.7.1. The Contact Definition 17
2.7.2. Disallowed keywords 17
2.7.3. LS-DYNA® Simulation 17
2.8. Global constraints 18
2.9. Setting up the Problem 18
2.9.1. The Information Panel 18
2.9.2. The Cases Panel 19
2.9.3. The Constraints Panel 20
2.9.4. The Parts Panel 22
2.9.5. Part Geometry 24
2.9.6. The Completion Panel 24
2.9.7. The Run Panel 25
2.9.8. The View Panel 26
2.10. Databases and Files 28

2.11. Opening and Saving Projects 28
2.12. Script Commands 28
3. Example Problems 29
3.1. Fixed Beam with Central Load 29
3.1.1. Problem Description 29
3.1.2. Input 29
3.1.3. Output 30
a) Convergence History 30
b) Density Contours 30
3.2. Beam using geometry definitions 31
3.2.1. Input 32
3.2.2. Output 32
a) Extrusion and Casting 32
b) Extrusion and two-sided casting 32
3.3. Multiple Load Cases 33
3.3.1. Problem Definition 33
3.3.2. Input 33
3.3.3. Output 34
a) Convergence History 34
b) Density Contours 34
3.4. Force-Displacement Constraints 35
3.4.1. Problem Definition 35
3.4.2. Input 35
3.4.3. Output 36
a) Convergence History 36
b) Density Contours 37
3.5. Linear Static Loading 37
3.5.1. Problem Definition 37
3.5.2. Input 38
3.5.3. Output 38
a) Convergence History 38
b) Density Contours 38
3.6. Shell Example 39
3.6.1. Problem Definition 40
3.6.2. Input 40
3.6.3. Output 40
a) Convergence History 40
b) Final Shell Thicknesses 41
4. Troubleshooting 42
4.1. Wrong Executable 42
4.2. Design Part 42
4.3. Extrusion Set 42
4.4. Negative Volumes 42
4.5. The LS-DYNA analysis fails if a smaller mass fraction is requested 42
4.6. Convergence 43
4.7. LS-PREPOST 43

4.8. Casting definitions ... 43
4.9. Mysterious Error when/after calling LS-DYNA and/or Errors involving the LSOPT Environment Variable ... 43
5. Appendix A: Scripting ... 44
5.1. The scripting language ... 44
5.2. Code Execution ... 44
5.3. Data-structures ... 44
5.3.1. lst_Root ... 44
5.3.2. lst_Method ... 44
5.3.3. lst_Problem ... 45
5.3.4. lst_Part ... 45
5.3.5. lst_Geometry ... 46
5.3.6. lst_Case ... 47
5.3.7. lst_Constraint ... 48
5.3.8. lst_JobInfo ... 48
5.4. Interactions with the Data Structures ... 49
5.4.1. Definition ... 49
5.4.2. Initialization ... 49
a) Adding Case Data ... 49
b) Accessing a Specific Case Structure ... 49
c) Adding Constraints ... 50
d) Adding Part Data ... 50
e) Accessing a Part ... 51
f) Adding Geometry Data ... 51
g) Adding Job Distribution Data ... 51
h) Specifying Optimization Method Parameters ... 52
5.4.3. Execution Functions ... 52
a) Saving the Project Data ... 52
b) Reading the Project Data ... 52
c) Create Topology ... 53
d) Cleaning the directory ... 53
5.5. Accessing Results ... 53
5.6. Example Script ... 54
5.6.1. Retrieving a value from the project database ... 54
5.6.2. Restart for an additional iteration ... 54
5.6.3. Creating a topology database ... 55
5.6.4. Printing the content of the project database ... 55
5.6.5. Printing the content of the results database ... 57
6. Appendix B: Theory ... 58
6.1. Background ... 58
6.2. Implementation ... 58
6.2.1. Definition ... 58
6.2.2. Creating the variables ... 59
6.2.3. Filtering of results ... 59
6.2.4. Material Parameterization ... 59
6.2.5. Design Objectives and Constraints ... 60

6.2.6. Initialization of the Lattice 60
6.2.7. Simulation to Obtain Field Variables 60
6.2.8. Constraint Handling 61
6.2.9. State Update Rules 61
6.2.10. Stopping Criteria 62
6.3. References 62
7. Appendix C: Using a queuing system 64
7.1. Relationship with the LS-OPT queuing system 64
7.2. Experience may be required 64
7.3. Introduction 64
7.4. Installation 64
a) Installation for all remote machines running LS-DYNA 65
b) Installation on the local machine 65
7.5. Example 65
7.6. Mechanics of the queuing process 67
7.7. Environment variables 68
7.8. Troubleshooting 68
7.9. User-defined queuing systems 69
7.10. Blackbox queueing system 69
7.11. Honda queuing system 72
7.12. Microsoft Windows Compute Cluster server 74
7.13. Simple manual setup for running LS-OPT and solvers on different machines 74
7.14. Passing environment variables 75
7.14.1. Adding a new environment variable definition 75
7.14.2. Editing an existing environment variable definition 75
7.14.3. Set by browsing 76
7.14.4. Edit browse list 77
7.14.5. How the browse list is used by LS-TaSC 78

TABLE OF FIGURES

Figure 1-1: Geometry definitions. 15
Figure 1-2: The faces selected for design in a casting definition are all the faces facing the material removal direction. The extra faces will cause a problem for the algorithm. 16
Figure 1-3: The information panel. 19
Figure 1-4: The cases panel. 20
Figure 1-5: The constraints overview panel. 21
Figure 1-6: The constraints creation panel. 21
Figure 1-7: The parts panel. 23
Figure 1-8: The panel to create part and geometry. 23
Figure 1-9: Creating a geometry constraint. 24
Figure 1-10: The method panel. 25
Figure 1-11: The run panel. 26
Figure 1-12: The view panel. 27
Figure 1-13: Viewing the model evolution in LS-PREPOST. 27
Figure 3-1: Geometry and loading condition of a single-load case example. 29
Figure 3-2: Convergence history of the total IED and change in design variables. 30
Figure 3-3: Initial and final density contours. 31
Figure 3-4: Evolution of the geometry shown using density contours. 31
Figure 3-5: Evolution of the beam using extrusion and single-sided casting constraints. 32
Figure 3-6: Evolution of the beam using extrusion and two-sided casting constraints. 33
Figure 3-7: The geometry and loading conditions of the multiple load case example. 33
Figure 3-8: Convergence history for multiple-load case example. 34
Figure 3-9: Initial and final density contours. 34
Figure 3-10: Evolution of the geometry for multiple-load case structure. 35
Figure 3-11: The geometry and loading conditions of the multiple constraints example. 35
Figure 3-12: Convergence history for the example with multiple constraints. 36
Figure 3-13: Evolution of the geometry for multiple-constrained clamped beam. 37
Figure 3-14: The geometry and loading conditions of a statically loaded structure. 37
Figure 3-15: Convergence history for linear-static example. 38
Figure 3-16: Initial and final density contours. 39
Figure 3-17: Evolution of the geometry for statically loaded structure. 39
Figure 3-18: The geometry and loading conditions of the shell example. The left side is built-in, while a downward load is applied to the right, back corner. 40
Figure 3-19: Convergence history for the shell example. 40
Figure 3-20: Final geometry and thicknesses for the shell problem. 41
Figure 6-1: The HCA algorithm for structural optimization. 58
Figure 6-2: Design variable update. 62

1. INTRODUCTION

1.1. *Classification of Structural Optimization Techniques*

Engineering optimization finds new designs that satisfy the system specifications at a minimal cost. Different types of structural optimization are:

1.1.1. Topology Optimization

This is a first-principle based approach to develop optimal designs. In this method, the user needs to provide the design domain, load and boundary conditions only. The optimal shape including the shape, size, and location of gaps in the domain is derived by the optimizer. While the most flexible method, topology optimization is indeed the most complex optimization method due to a multitude of reasons, like, large number of design variables, ill-posed nature of the problem, etc. Nevertheless, the benefits of using topology optimization include the possibility of finding new concept designs that have become feasible due to recent advances in technology, e.g., new materials. The LS-aSC program can be used to this design work.

1.1.2. Topometry Optimization

Topometry optimization, a methodology closely related to topology optimization, changes the element properties on an element by element basis. With the LS-TaSC program, the shell thicknesses can be designed.

1.1.3. Size Optimization

In this mode, the designer has already finalized the configuration of the system but improvements are sought by changing the thickness of members of the structure on a part basis instead of an element by element basis as done for topometry optimization. There is usually no need to re-mesh the geometry. This class of optimization problems is the most amenable to meta-model based optimization. The LS-OPT® program should be used for this instead of this program.

1.1.4. Shape Optimization

Shape optimization further expands the scope of design domain by allowing changes in the geometry of the structure, for example the radius of a hole. While there is more freedom to explore the design space, the cost of optimization increases due to the possible need to mesh different candidate optimum designs. Use the LS-OPT® program together with a preprocessor such as LS-PREPOST® instead of this program.

1.2. *Brief Overview*

Topology optimization in structures has been studied since the 1970s resulting in many books and numerous papers. The books by Rozvany [1] and Bendsøe and Sigmund [2] provide a very comprehensive and contemporary survey of optimization techniques used in topology optimization. Most previous studies [3, 4] in topology optimization have focused on designing structures with static loading conditions but there is relatively little work on handling problems involving dynamic loads, like those observed in crashworthiness optimization [5]. The topology optimization in the context of crashworthiness is a very complex problem due to non-linear interactions among material non-linearities, geometry, and transient nature of boundary conditions.

The most efficient topology optimization methods use sensitivity information (optimality criterion based methods, Rozvany [1], Bendsøe and Kikuchi [6]) to drive the search for an optimum. Sensitivity calculations are computationally inexpensive for linear-static problems but not for the problems that involve non-linearities. To use the same set of topology optimization methods, one needs to explicitly calculate sensitivities which is practically infeasible due to very high computational cost involved with simulations. Thus the theory used to solve the linear-static load cases, though quite mature, is not practical for the crashworthiness problems and alternate methods need to be explored.
Previously different approaches have been adopted by authors to solve topology optimization with nonlinearities. Pedersen used the Method of Moving Asymptotes for crashworthiness optimization of two-dimension structures [7]. They used a quasi-static nonlinear FEA to account for geometric nonlinearities to handle large deformation and rotation of plastic beam elements. However, the method ignored the contact between elements arising due to nonlinear behavior of the structures. Soto [8, 9] presented a heuristics based method using a prescribed plastic strain or stress criterion to vary the density to achieve the desired stress or strains with a constraint on mass. However, this method could not be generalized to solid structures. Pedersen [10] used beam elements to handle topology in crashworthiness optimization. Forsberg and Nilsson [11] proposed two algorithms to get a uniform distribution of the internal energy density in the structure. In the first method, they deleted inefficient elements and in the second method they updated the thicknesses of the shell elements. This method also was limited to a small set of optimization problems. Shin et al. [12] proposed an equivalent static load method where they calculated an equivalent static load for the dynamic problem and then used the linear-static topology optimization techniques to find the optimal topology. The main difficulty in this method is the requirement to accurately compute the equivalent loads.

1.3. Topology Optimization Method in LS-TaSC

A heuristic topology optimization method developed at the University of Notre Dame, known as hybrid cellular automata [13], showed potential in handling topology optimization problem for crashworthiness problems. This method updates the density of elements based on the information from its neighbors. No gradient information was required. The simplicity and effectiveness of this method for both two- and three-dimensional problems made it an attractive choice for our initial implementation. The methodology has however been enhanced using more established approaches as well; currently, amongst others, it gives mesh independent results.

This manual is divided into parts. The user's manual describes how to do topology optimization using LS-TaSC. A few examples are provided to cover different options in the topology optimization program. The scripting section lists the command language used to interact with the topology optimization code together with some examples. Some common errors and tips on troubleshooting are provided in a separate chapter. In the theory section, the method for topology optimization is described. Setting up queuing systems is described in an appendix.

1.4. References

1. GIN Rozvany, *Structural Design via Optimality Criteria*, Kluwer, London, 1989.
2. MP Bendsøe, O Sigmund, *Topology Optimization: Theory, Methods and Applications*, Springer-Verlag, Heidelberg, 2003.
3. HA Eschenaur, N Olhoff, Topology Optimization of Continuum Structures: A Review, *Applied Mechanics Review*, 54(4), 331-390, 2001.
4. GIN Rozvany, *Topology Optimization in Structural Mechanics*, Springer-Verlag, Vienna, 1997.
5. CA Soto, Applications of Structural Topology Optimization in the Automotive Industry: Past, Present, and Future, in HA Mang, FG Rammerstorfer, J Eberhardsteiner (eds), *Proceedings of the Fifth World Congress on Computational Mechanics*, Vienna, 2002.
6. MP Bendsoe, N Kikuchi, Generating Optimal Topologies in Optimal Design using a Homogenization Method, *Computer Methods in Applied Mechanics and Engineering*, 71(2), 197-224, 1988.
7. CBW Pedersen, Topology Optimization Design of Crushed 2d-Frames for Desired Energy Absorption, *Structural and Multidisciplinary Optimization*, 25, 368-282, 2003.
8. CA Soto, Structural topology optimization: from minimizing compliance to maximizing energy absorption, *International Journal of Vehicle Design*, 25(1/2), 142-163, 2001.
9. CA Soto, Structural Topology Optimization for Crashworthiness, *International Journal of Numerical Methods in Engineering*, 9(3), 277-283, 2004.
10. CBW Pedersen, Crashworthiness Design of Transient Frame Structures Using Topology Optimization, *Computer Methods in Applied Mechanics in Engineering*, 193, 653-678, 2004.
11. J Forsberg, L Nilsson, Topology Optimization in Crashworthiness Design, *Structural and Multidisciplinary Optimization*, 33, 1-12, 2007.
12. MK Shin, KJ Park, GJ Park, Optimization of Structures with Nonlinear Behavior Using Equivalent Loads", *Computer Methods in Applied Mechanics and Engineering*, 196, 1154-1167, 2007.
13. A Tovar, *Bone Remodeling as a Hybrid Cellular Automaton Optimization Process*, PhD Thesis, University of Notre Dame, 2004.

2. USER'S MANUAL

Topology optimization consists of describing the topology design problem together with the solution methodology, the scheduling the automated design, and the evaluation of the results.

2.1. Running the Program

The LS-TaSC GUI is launched from the command prompt by running the executable (*lstasc*). If a project already exists, then the project database name (**.lstasc*) can be supplied in two ways:

1. With the execution command
   ```
   $ lstasc myProject.lstasc
   ```
2. The *file open* dialogue, available from the File pulldown menu

2.2. Design Goal

The goal of topology optimization is to find the shape of a structure with the maximum utility of the material. For dynamic problems like crashworthiness simulations, this is achieved by designing for a uniform internal energy density in the structure while keeping the mass constrained.

2.3. Problem Definition

The topology design problem is defined by (i) the allowable geometric domain, (ii) how the part will be used, and (iii) properties of the part such as manufacturing constraints. Additionally, you have to specify methodology requirements such as termination criteria and management of the LS-DYNA® evaluations. In the GUI, provide this information using the following headings:

- *Cases* These store the load case data such as, the LS-DYNA® input deck and executable to use. The *Cases* data therefore contain the information on how to simulate the use of part.
- *Parts* The properties of the parts such as the part ID, mass reduction, and geometric definitions are given here.
- *Constraints* This optional information prescribes the stiffness or compliance of the whole structure.
- *Completion* These are methodology data such as the convergence criterions.

2.4. The Design Parts

The design domain is specified by selecting parts – the optimum parts computed will be inside the boundaries delimited by these parts. The part must be defined using *PART, not *PART_*OPTION*. The parts may contain holes: a structured mesh is accordingly not required.

2.4.1. Elementwise Material Density and Element Deletion for Solids

The shape of a solid part is described by the subset of the initial elements used. The shape of a solid element is controlled by changing the amount of material in the element. This is achieved by assigning a design variable to the density of each element. The material is parameterized using a so-called *density approach*. In this approach, a design variable is directly linked to the individual material element such that each cell has its own material model. The design variable x, also known as relative density, varies from 0 to 1 where 0 indicates void and 1 represents the full material. The material properties corresponding to the values of design variables are obtained using an appropriate interpolation model as described in the theoretical manual. The upper bound on the design variable is 1, while elements with design variable value less than a user-defined minimum value (0.05 for dynamic problems, and 0.001 for linear) are deleted to improve numerical stability.

2.4.2. Design of Shells

For shells the thickness are changed to achieve a uniform internal energy density in the part. The upper bound on the design variable is the original shell thickness, while elements with design thickness values less than a user-defined minimum value (0.05 for dynamic problems, and 0.001 for linear) are deleted to improve numerical stability.

2.4.3. Element types

Solid elements must be eight-noded solid elements or four-noded tetrahedral elements. Elements shapes close to perfectly cubic are the best for the current neighbor selection algorithm.

Shell elements may be four-noded shell elements or three-noded shell elements. The triangular elements must be specified as four-noded shell elements by specifying the last node twice. Elements shapes close to perfectly square or an equilateral triangle are the best for the current neighbor selection algorithm.

Tetrahedral and triangular elements cannot be extruded.

2.4.4. Material data

The part must be modeled using *MAT_PIECEWISE_LINEAR_PLASTICITY.

The load curve option (LCSS) is not supported; use the EPS*i*/ES*i* variables. Test the material using LS-DYNA before using it in LS-TaSC. For some material data the topology algorithm (SIMP algorithm) can only create materials for which the slope of the stress-strain curve is higher in plastic regime than in the elastic one; in this case the errors and warnings should be consulted for feedback on how to modify the material stress-strain curve in the input deck.

2.5. *Geometry and manufacturing definitions*

For each part geometry and manufacturing constraints such as being an extrusion may be specified.

The geometry definitions, as shown in Figure 2-1, are:

- *Symmetry* For these the geometry is duplicated across a symmetry plane. The part as supplied by the user must be symmetric: an element must have a matching element on the other side of the symmetry plane.
- *Extrusion* An element set is extruded in a certain direction. Allowable set definitions are *SET_SOLID, *SET_SOLID_LIST, *SET_SHELL, and *SET_SHELL_LIST. The part as supplied by the user must be an extrusion with every element in the elements set must have the same number of extruded elements. Only hexahedrons and quadrilateral elements can be extruded.
- *Casting* Material is removed only from a given side of the structure. The structure therefore will have no internal holes. The casting constraints can be one sided or two-sided. This capability is available only for solids.

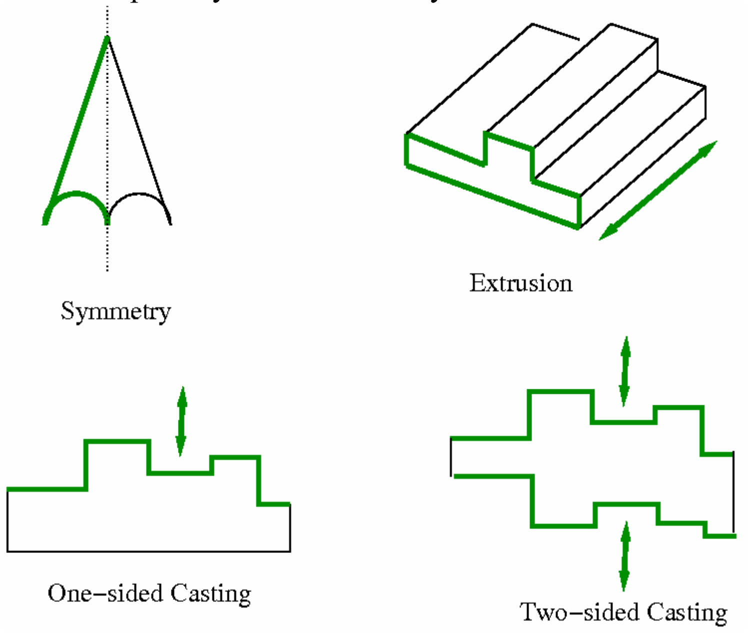


Figure 2-1: Geometry definitions

Multiple geometry constraints can be specified for each part. Some combinations of geometry constraints may however not be possible. A maximum of three geometry definitions per part is possible. The symmetry planes must be orthogonal to each other, the extrusion direction must be on the symmetry planes, the casting direction must be on the symmetry planes, and the extrusion directions must be orthogonal to casting directions. Only one casting definition may be defined per part.

The symmetry and extrusion definitions are implemented by assigning multiple elements to a variable, while the casting definitions are implemented as inequality constraints requiring certain variables to be larger than others according to the cast direction.

For a casting definition, the free faces are selected as shown in Figure 2-2. It can be seen that the algorithm will select faces inside a hole. All of the material shown can be considered to be defined using a single *PART definition, from which it can be noted that the object to the right is considered for design even though it is in the ‘shadow’ of the object to the left. These extra faces will cause the algorithm to fail. An analyst can enforce a desired behavior by breaking the part up in smaller parts and applying the casting definition only where desired.

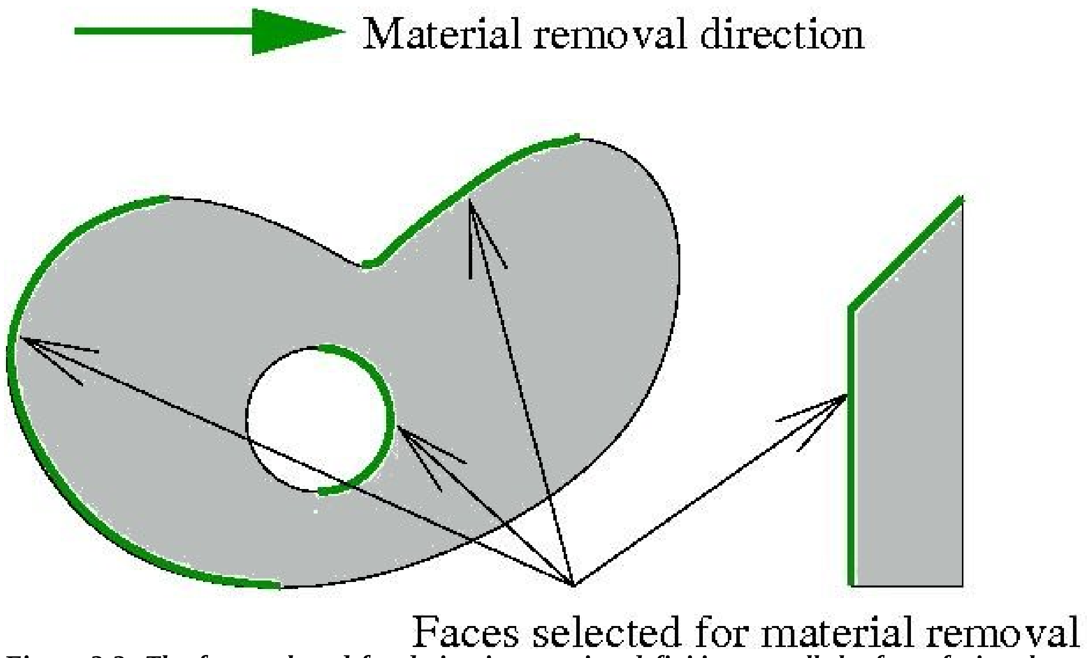

Material removal direction
Faces selected for material removal

Figure 2-2: The faces selected for design in a casting definition are all the faces facing the material removal direction. The extra faces will cause the algorithm to fail.

2.6. Design Variables

2.6.1. Mapping Elements to the Design Variables

A design variable is assigned to every finite element in the design parts. For geometry constraints, the variables are defined only on a subset of elements.

2.6.2. Filtering of Results

Structured grids are not always possible for industrial applications, and the results should be mesh independent. A radius based strategy is therefore used to identify neighbors. In this strategy, a virtual sphere (of default or user-defined radius) is placed at the center of an element. All elements that are within this sphere are considered the neighbors of the corresponding element. The result at an element is computed scaled from its own value and of its neighbors.

For dynamic problems, it was observed that accounting for the history of evolution induces stability by reducing the element deletion rate. Hence, the field variable (internal energy density) of i^{th} cell at iteration t is updated by defining a weighted sum on the field variable of three previous iterations.

2.6.3. Initialization of the Design Variables

The design variables are initialized to satisfy the mass fraction. All variables in a part are assigned the same initial value. All associated field variables are also initialized to zero.

2.7. LS-DYNA® Specifics

2.7.1. The Contact Definition

The contacts involving the design parts should be modeled using either *CONTACT_AUTOMATIC_SURFACE_TO_SURFACE[_ID] or *CONTACT_AUTOMATIC_SINGLE_SURFACE[_ID] options. These contact options are general enough to accommodate the changes in the geometry of the design parts during the optimization to maintain valid contacts. It is also recommended to specify the contact options (e.g., friction coefficients) appropriately accounting for the changes in the geometry may result in significantly different material properties for some elements near the contacts. Too restrictive values may cause instabilities in the LS-DYNA® simulations for intermediate geometries.

2.7.2. Disallowed keywords

The *INCLUDE keyword is not supported in the current version.

The portions of the FE model related to the design part are extensively edited by the optimization algorithm. In these segments of the FE model only specific versions of *PART, *SET, and *CONTACT keywords may be used as described in the relevant sections. Portions of the model not edited by the optimization algorithm are not subjected to this rule.

2.7.3. LS-DYNA® Simulation

The elements in the finite element model are modified by changing the material models, adding or deleting elements, at each iteration. The input deck is accordingly re-written for every iteration. The relevant field variables for all elements are obtained from the output to completely define the state of each cell. For multiple load case conditions, the state variable is based on the output from simulations of different load cases.

This modified input deck is analyzed using LS-DYNA®. One can take advantage of multiple processors using the MPP version of LS-DYNA®. Queuing system can also be used as described in Section 2.9.2.

The internal densities of the cells are extracted at the end of the analysis for use in the design procedure.

2.8. *Global constraints*

Global responses depend on the design of the whole structure. Two types of global responses are:

- Stiffness. This is specified as displacement constraint.
- Compliance. This is specified as a reaction force constraint.

Local effects such as stress concentrations are not handled by this algorithm.

The algorithm is actually a search for the mass of the structure. If the displacements are too large, then mass are added to the structure to increase the stiffness. If the reaction forces are too large, then mass is removed from the structure to reduce the force.

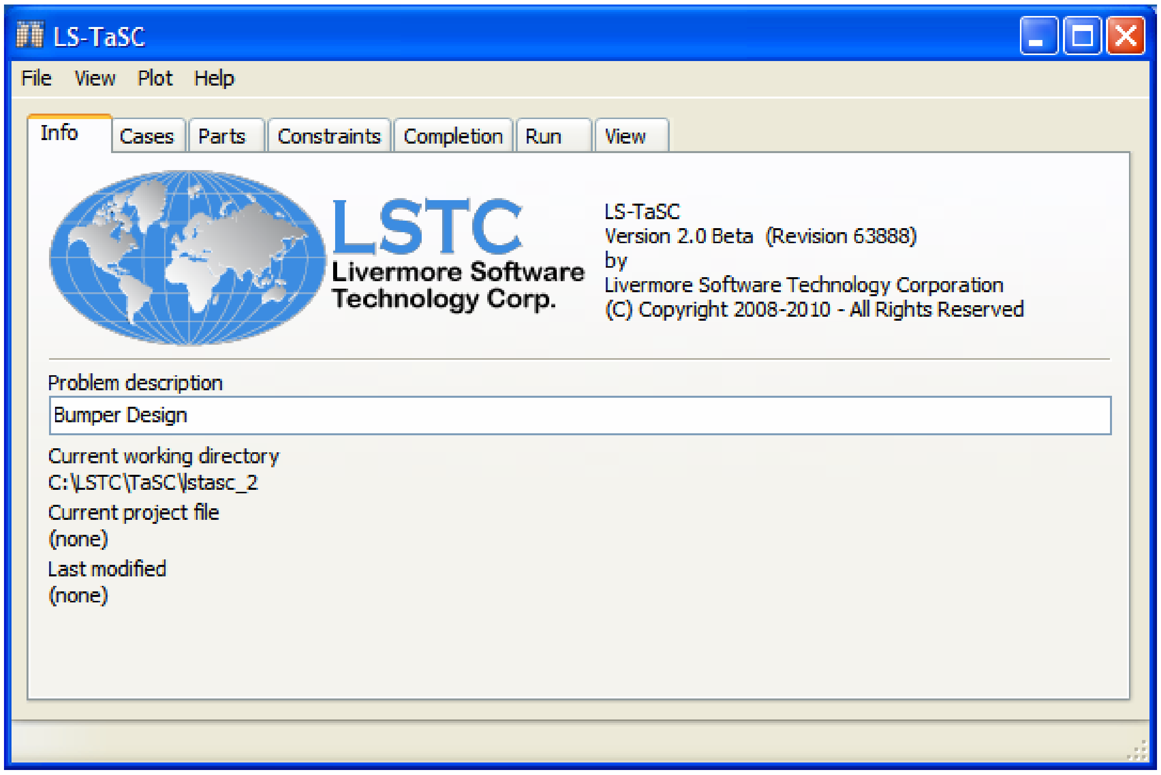
Multiple global constraints may be specified. If the constraints are in conflict, then a trade-off is done, and a design is selected resulting in the minimum violation of any given constraint.

2.9. *Setting up the Problem*

The GUI consists of a number of panels. Complete the panels from left to right as described in the following subsections.

2.9.1. The Information Panel

The information contains only information such as the software version, the name of the current database file, and a description of the problem.


LS-TaSC
File View Plot Help
Info Cases Parts Constraints Completion Run View
LSTC Livermore Software Technology Corp.

LS-TaSC
Version 2.0 Beta (Revision 63888)
by
Livermore Software Technology Corporation
(C) Copyright 2008-2010 - All Rights Reserved

Problem description
Bumper Design
Current working directory
C:\LSTC\TaSC\lstasc_2
Current project file
(none)
Last modified
(none)

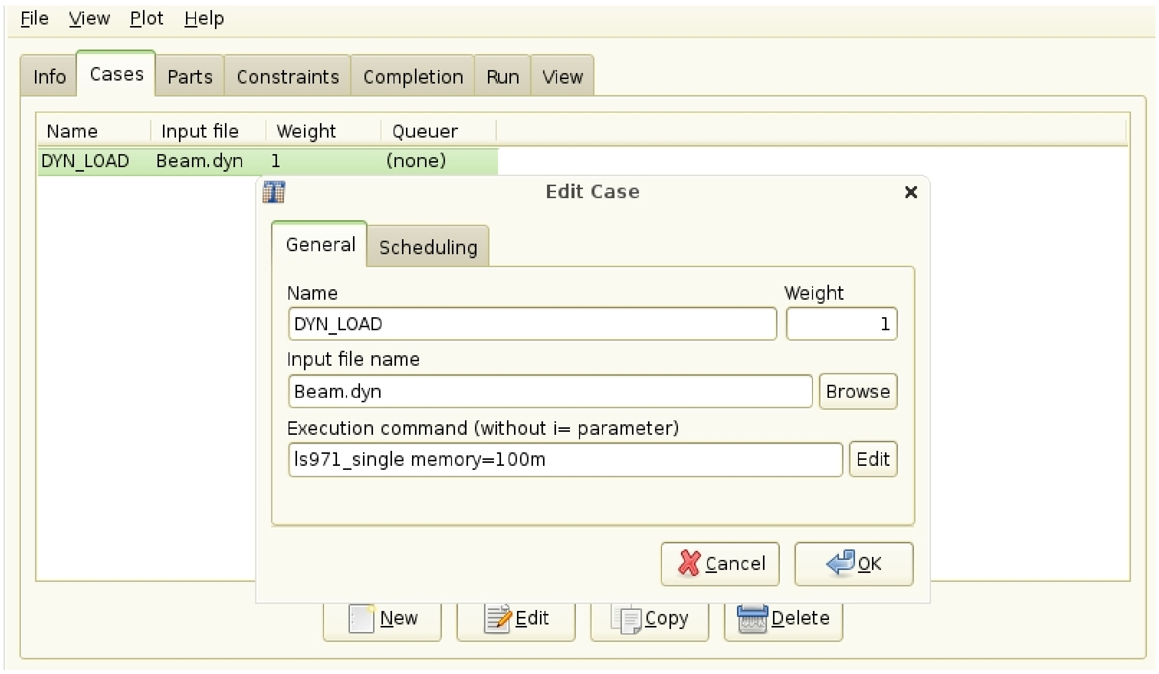
Figure 2-3: The information panel.

2.9.2. The Cases Panel

The cases panel contains all of the load cases to be analyzed using LS-DYNA®. See the following table and Figure 2-4 for more details.

Cases data	
`Name`	Each case is identified with a unique name e.g., TRUCK. The same name would be used to create a directory to store all simulation data.
`Execution Command`	The complete solver command or script (e.g., complete path of LS-DYNA executable) is specified.
`Input File`	The LS-DYNA input deck path is provided.
`Weight`	The weight associated with a case is defined here. This enables the user to specify non-uniform importance while running multiple cases.
`Number of jobs`	This parameter indicates the number of processes to be run simultaneously. A value of zero indicates all processes would be run simultaneously. This parameter only makes sense if multiple cases must be evaluated. The program will allow as many processes as defined for the current case being evaluated.
`Queue system`	This parameter is used to indicate the queuing system. The options are: lsf, loadleveler, pbs, nqs, user, aqs, slurm, blackbox, msccp,

	pbspro, Honda. By default, no queuing system would be used. See the appendix for a description of setting up the queuing systems. The system is the same as used in LS-OPT®, so a queuing system definition is the same.



Name	Input file	Weight	Queuer
DYN_LOAD	Beam.dyn	1	(none)

Name: DYN_LOAD
Weight: 1
Input file name: Beam.dyn
Execution command (without i= parameter): ls971_single memory=100m

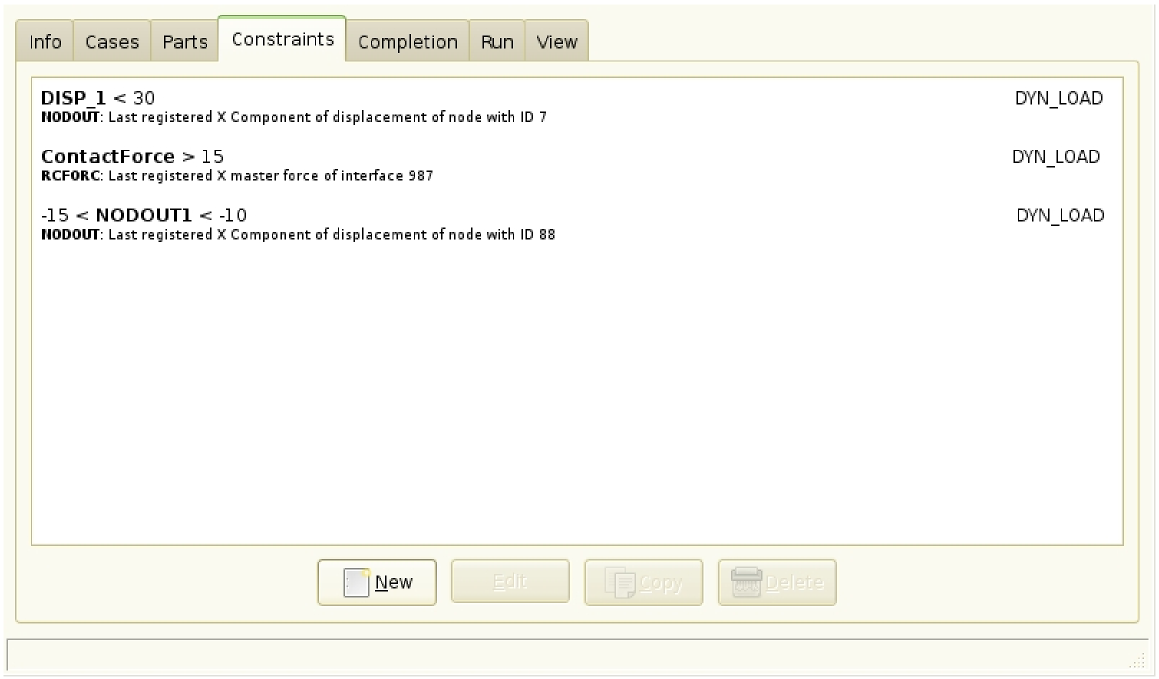
Figure 2-4: The cases panel.

2.9.3. The Constraints Panel

The constraint panel contains the global constraints on the structure. See the following table and Figure 2-4 for more details.

Cases data

`Name`	Each constraint is identified with a unique name e.g., MAX_DISP.
`Case`	Each constraint is associated with a load case.
`Constraint Type`	One of NODOUT or RCFORC
`Lower and upper bound`	The weight associated with a case is defined here. This enables the user to specify non-uniform importance while running multiple cases.
`ID`	This is the ID of the node in the FE model at which the results must be collected.
`Select`	This parameter indicates which value over time must be selected. It can be the last value, the maximum value, the minimum value, or at a specific time. A time, or a time interval can also be specified.
`Filtering`	If filtering is desired, select the type of filter, frequency, and time units. LS-PREPOST can be used to investigate the effects of filtering.

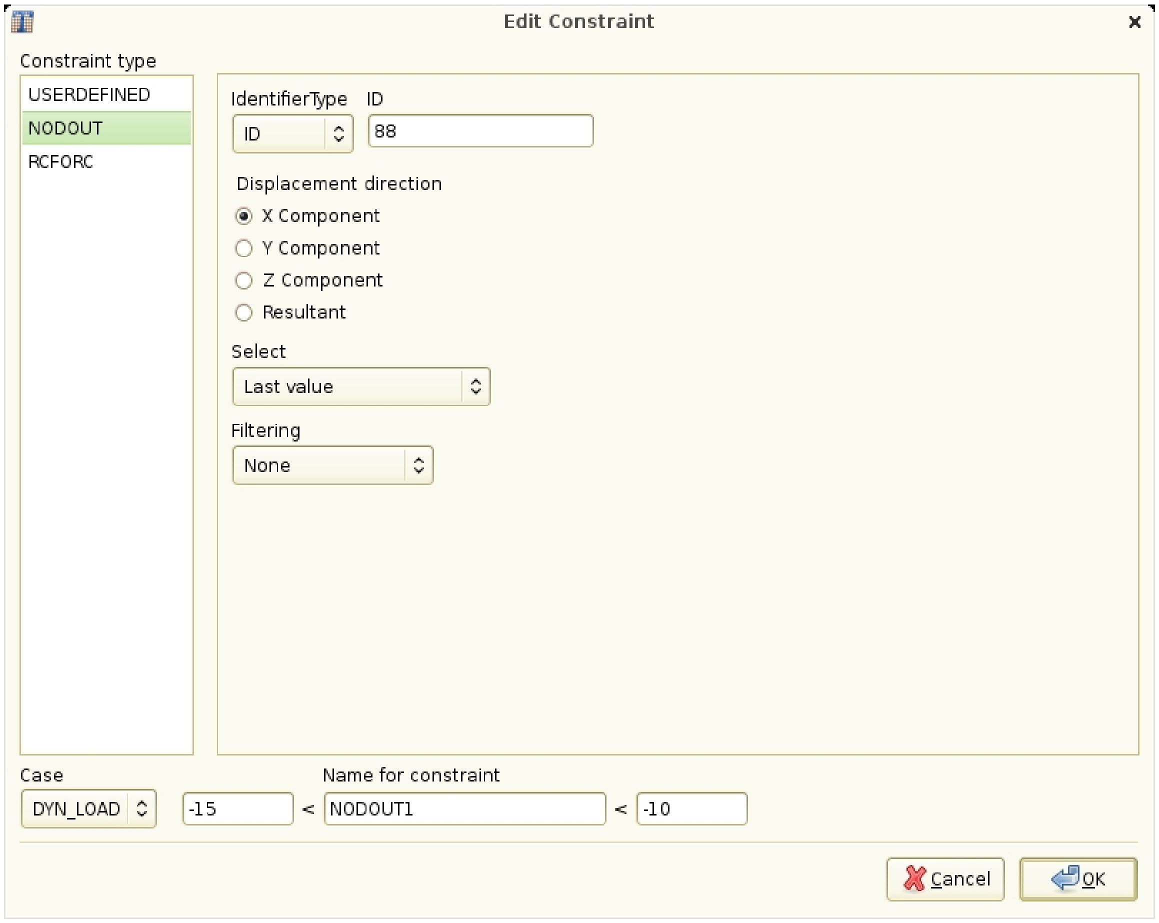


DISP_1 < 30 DYN_LOAD
NODOUT: Last registered X Component of displacement of node with ID 7

ContactForce > 15 DYN_LOAD
RCFORC: Last registered X master force of interface 987

-15 < **NODOUT1** < -10 DYN_LOAD
NODOUT: Last registered X Component of displacement of node with ID 88

Figure 2-5: The constraints overview panel.



Constraint type
USERDEFINED
NODOUT
RCFORC

IdentifierType: ID
ID: 88

Displacement direction
[● X Component]
○ Y Component
○ Z Component
○ Resultant

Select: Last value
Filtering: None

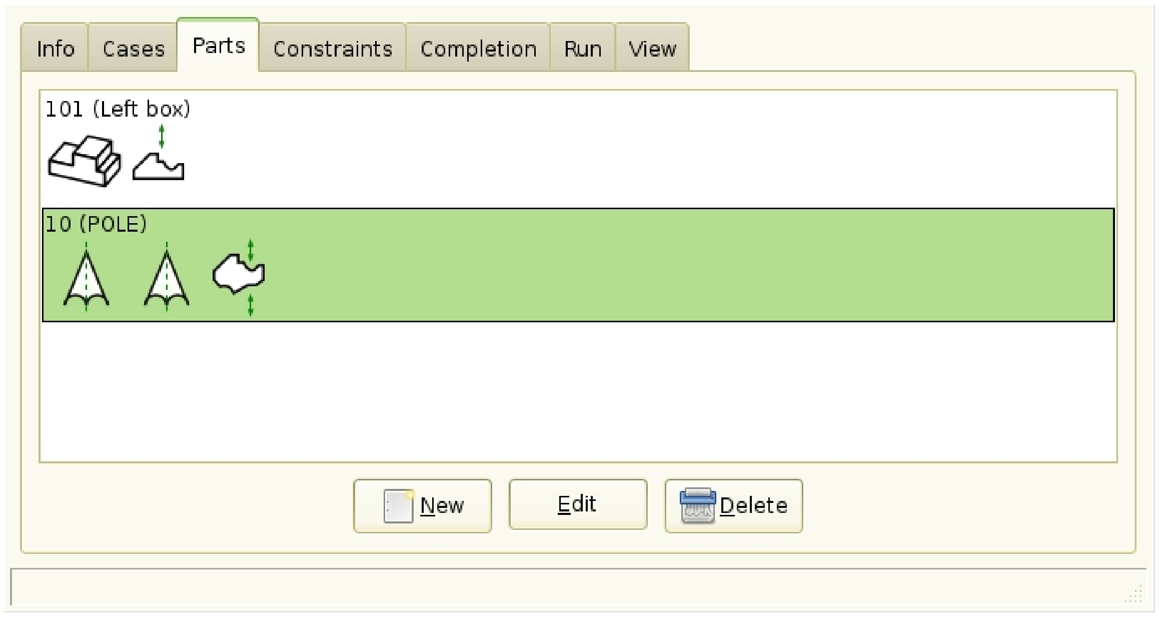
Case: DYN_LOAD
-15 < Name for constraint: NODOUT1 < -10

Figure 2-6: The constraints creation panel.

2.9.4. The Parts Panel

The part definition panel contains information about the parts to be designed, such as the geometry and mass fraction. See the following table, Figure 2-7 and Figure 2-8 for more details.

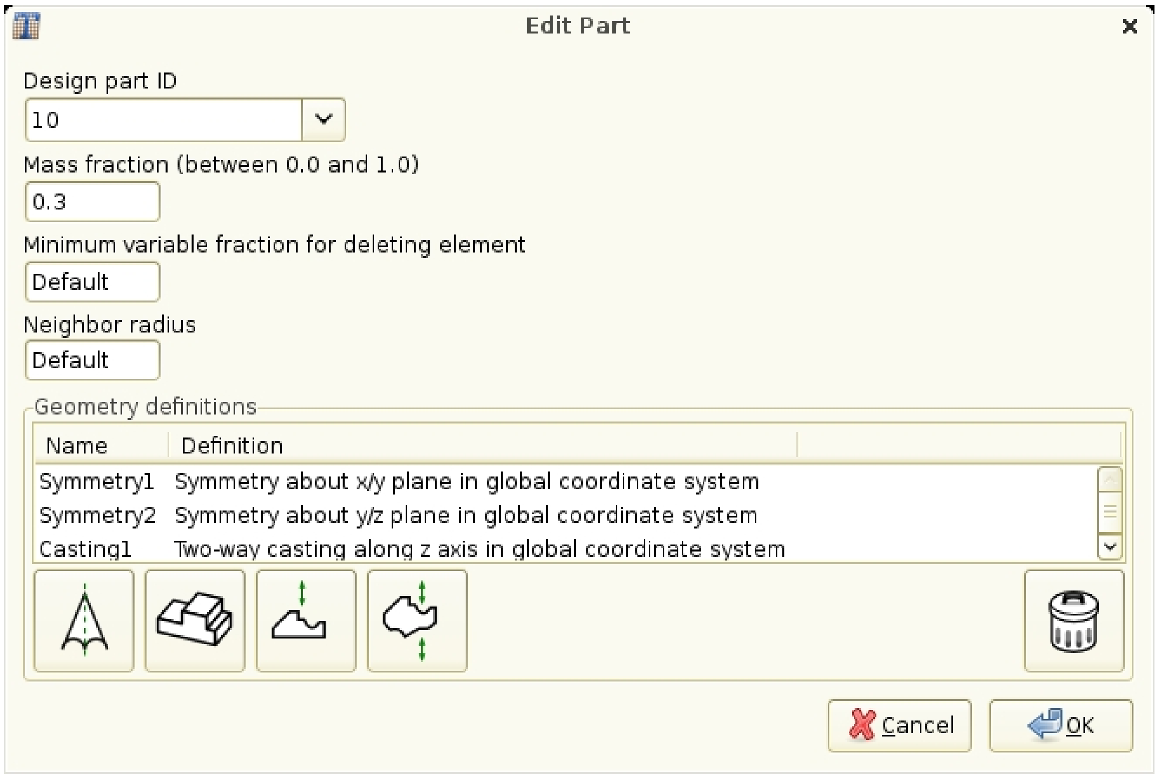
Part data	
`Design Part ID`	The user needs to specify the design domain for topology optimization. To facilitate the identification of design domain, all elements in the design domain are put in a single part in the LS-DYNA input deck. The information about the design domain is then communicated through the corresponding *part-id*. Note: For multiple load cases, the user must ensure that the design domain mesh and the *part-id* remain the same in all input decks.
`Mass Fraction`	This parameter describes the fraction of the mass of the part to be retained. The rest will be removed. A part with an initial weight of 5, designed using a *Mass Fraction* of 0.3 will have a final weight of 1.5.
`Neighbor Radius`	All elements within a sphere of radius of this value are considered the neighbors of an element. The design variable at an element is updated using the result at the element averaged together with that of its neighbors. Smaller values of this parameter yield finer-grained structures. The default value depends on the average element size.
`Minimum variable fraction`	If the design variable value associated with and elements is too small then that element is deleted to preserve the stability of the model. An appropriate value ($0.05 < x < 0.95$) is supplied here. The default is 0.05 for non-linear problems and 0.001 for linear problems.



101 (Left box)

10 (POLE)

Figure 2-7: The parts panel.



Design part ID
10

Mass fraction (between 0.0 and 1.0)
0.3

Minimum variable fraction for deleting element
Default

Neighbor radius
Default

Geometry definitions

Name	Definition
Symmetry1	Symmetry about x/y plane in global coordinate system
Symmetry2	Symmetry about y/z plane in global coordinate system
Casting1	Two-way casting along z axis in global coordinate system

Figure 2-8: The panel to create part and geometry.

2.9.5. Part Geometry

The geometric properties can be defined for every part. See the following table and Figure 2-9 for more details.

Geometry data	
`Name`	The geometric property can assigned a name or a default name can be used.
`Extrusion Set ID`	To define an extruded part, the user firstly creates a set of all solid elements that would be extruded (SET_SOLID). The *id* of this set is specified in the input deck to identify the extrusion set.
`Symmetry Plane`	Specify a symmetry plane to define symmetry.
`Cast direction`	A cast direction is required for a casting constraint. The direction can be negative. This is the direction in which the material will be removed. It is the opposite of the direction in which a casting die will be removed.
`Coordinate System ID`	The geometric property can be defined in a specific coordinate system or the default Cartesian system can be used.

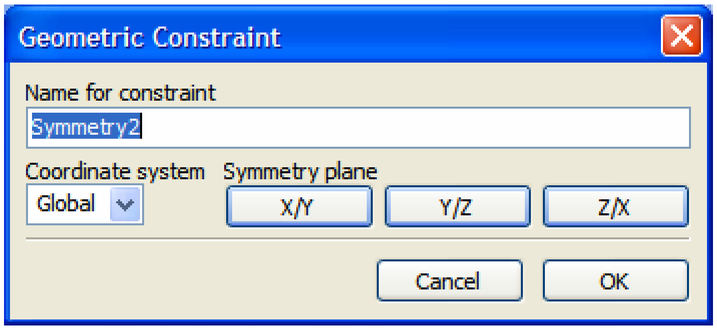
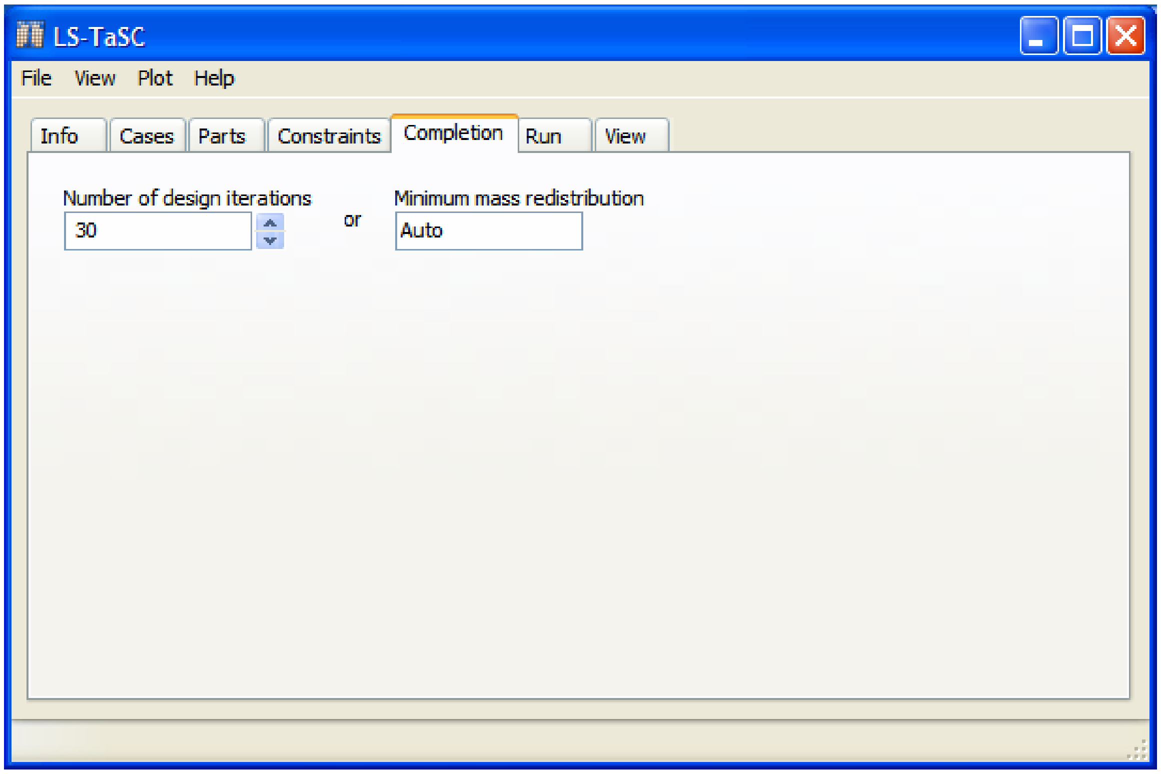


Figure 2-9: Creating a geometry constraint.

2.9.6. The Completion Panel

The completion panel specifies how the optimization problem will be solved. See the following table and Figure 2-10 for more details.

Completion data	
`Number of design iterations`	This is the maximum number of iterations allowed. The default value is 30.
`Minimum mass redistribution`	The minimum mass redistribution is the termination criterion used to stop the search when the topology has evolved sufficiently. This value is compared with the *Mass_Redistribution* history variable displayed in the view panel. The default value is 0.002.

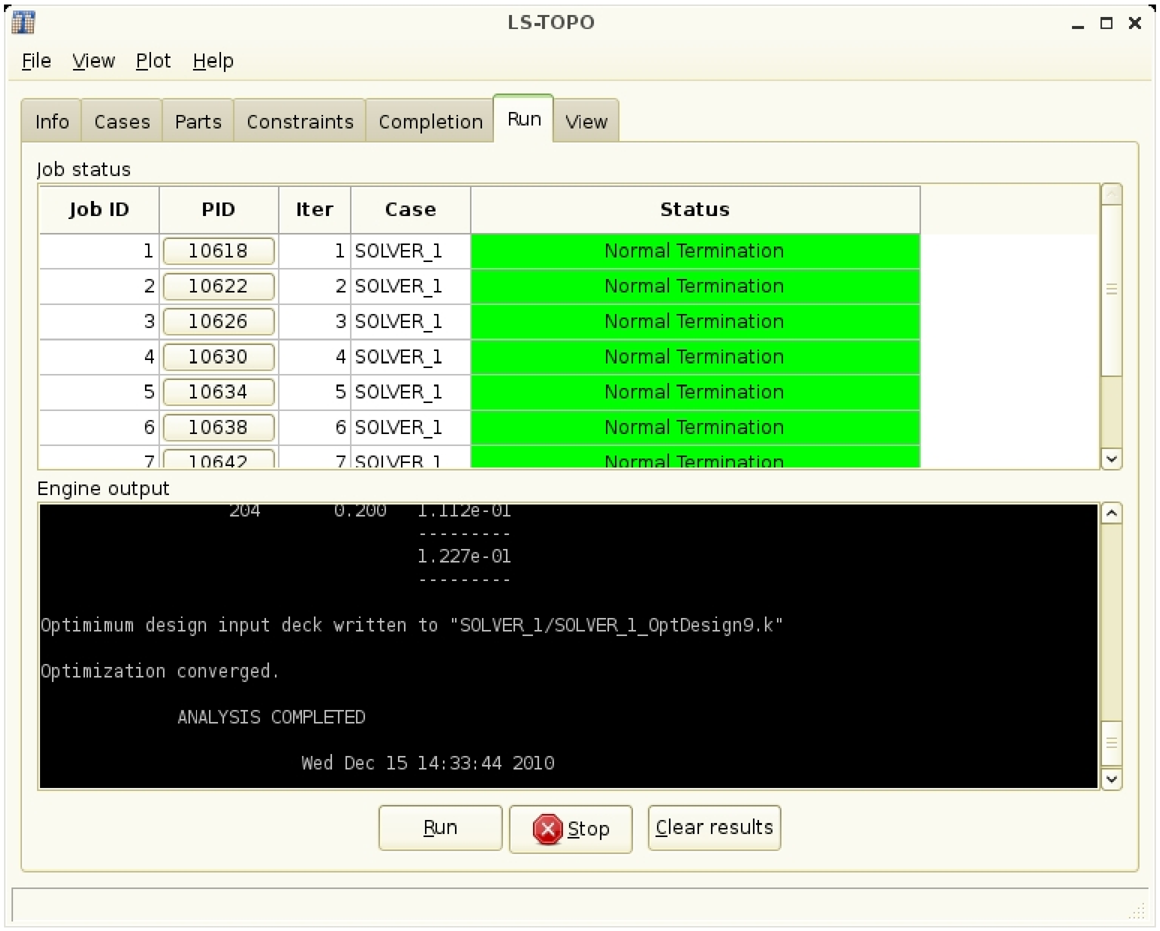


Number of design iterations: 30 or Minimum mass redistribution: Auto

Figure 2-10: The completion panel.

2.9.7. The Run Panel

The control panel is used to submit the design problem. In addition, the LS-DYNA® jobs can also be stopped, and old results deleted. Use this panel and the Viewer panel to monitor job execution. See Figure 2-11 for more details.



Job ID	PID	Iter	Case	Status
1	10618	1	SOLVER_1	Normal Termination
2	10622	2	SOLVER_1	Normal Termination
3	10626	3	SOLVER_1	Normal Termination
4	10630	4	SOLVER_1	Normal Termination
5	10634	5	SOLVER_1	Normal Termination
6	10638	6	SOLVER_1	Normal Termination
7	10642	7	SOLVER_1	Normal Termination

```
              204       0.200   1.112e-01
                                ---------
                                1.227e-01
                                ---------

Optimimum design input deck written to "SOLVER_1/SOLVER_1_OptDesign9.k"

Optimization converged.

          ANALYSIS COMPLETED

                 Wed Dec 15 14:33:44 2010
```

Figure 2-11: The run panel.

2.9.8. The View Panel

The view can be used to monitor both optimization progress and optimization results. Both histories and plots in LS-PREPOST are possible. See Figure 2-12 and Figure 2-13 for more details.

For the histories note that:

- Multiple histories can be plotted simultaneously by holding down the Control key.
- The plot ranges can be set under the View pulldown menu.
- The histories can be printed or saved to file using the Plot pulldown menu.
- The history data can be exported and postprocessed using the scripting interface.

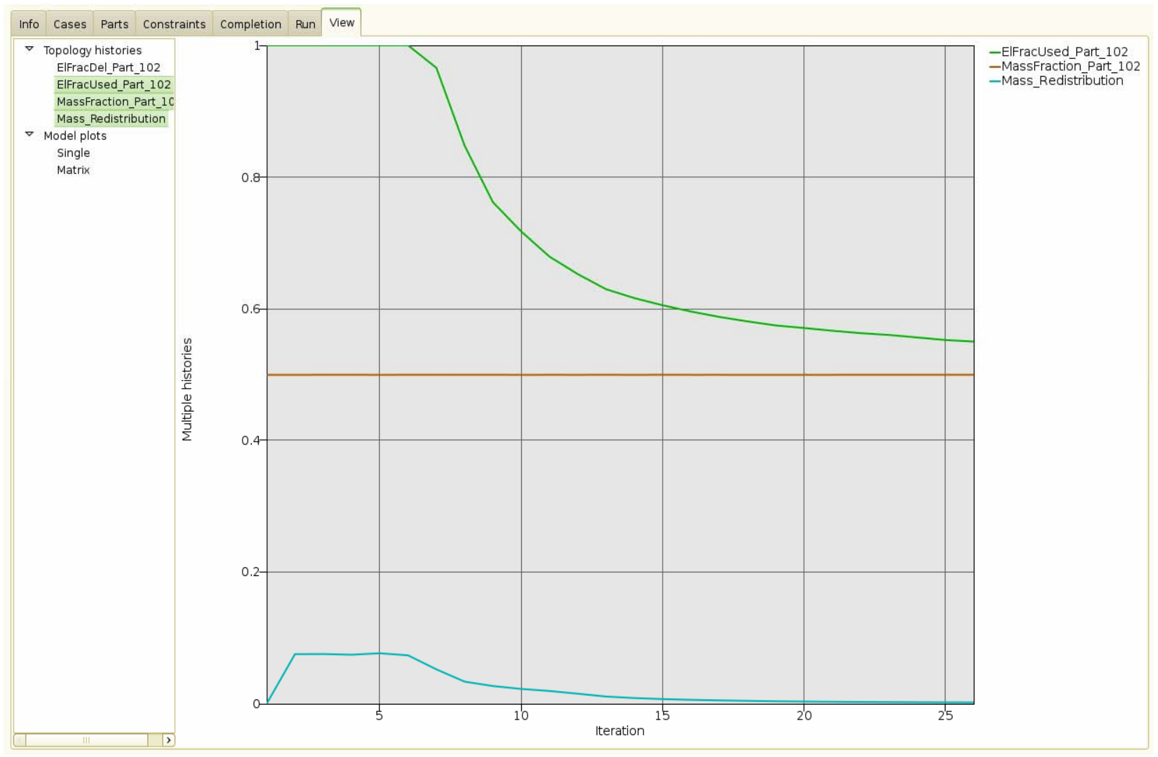
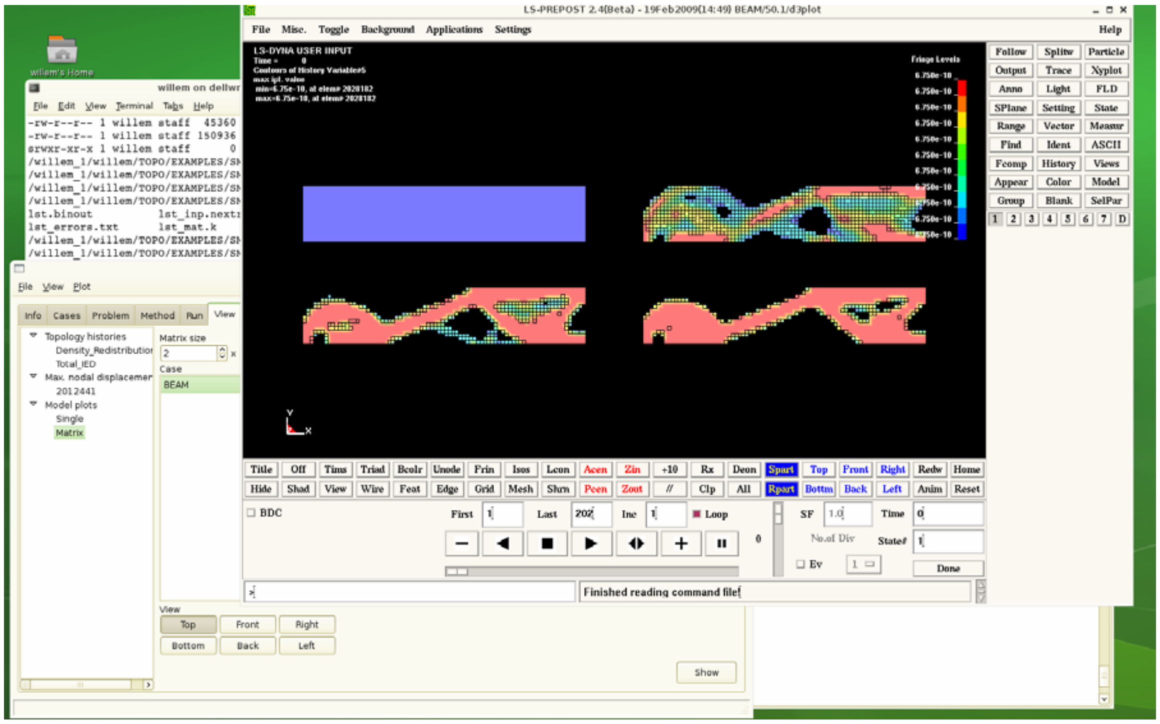


Figure 2-12: The view panel with histories.



```
-rw-r--r-- 1 willem staff  45360
-rw-r--r-- 1 willem staff 150936
srwxr-xr-x 1 willem staff      0
/willem_1/willem/TOPO/EXAMPLES/SM
/willem_1/willem/TOPO/EXAMPLES/SM
/willem_1/willem/TOPO/EXAMPLES/SM
/willem_1/willem/TOPO/EXAMPLES/SM
1st.binout             1st_inp.nextr
1st_errors.txt         1st_mat.k
/willem_1/willem/TOPO/EXAMPLES/SM
/willem_1/willem/TOPO/EXAMPLES/SM
```

```
LS-DYNA USER INPUT
Time =        0
Contours of History Variable#5
max ipt. value
 min=6.75e-10, at elem# 2028182
 max=6.75e-10, at elem# 2028182
```

Matrix size: 2 x

First: 1 Last: 202 Inc: 1 SF: 1.0 Time: 0 State#: 1

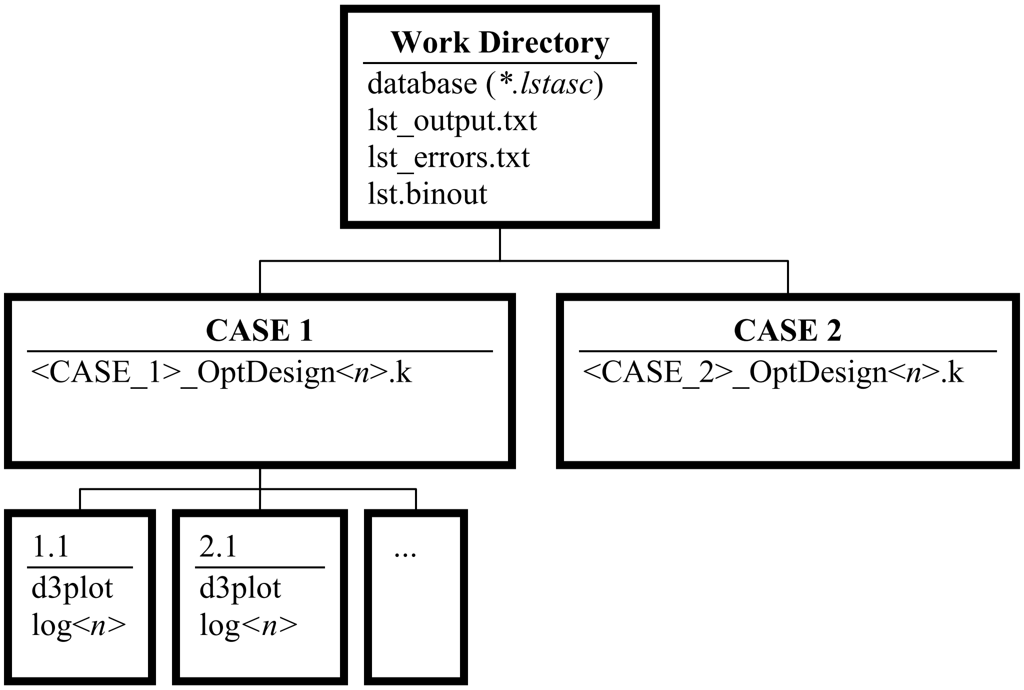
Finished reading command file!

Figure 2-13: Viewing the model evolution in LS-PREPOST.

2.10. Databases and Files

The important files and directories are shown in the figure below. Four files are important to know about:

- The project database
- The project results in the *lst.binout* binary file
- The optimal design in the case directory
- The d3plot files in the run directory inside the case directory



2.11. Opening and Saving Projects

The standard File pulldown is provides the ability to open and save projects. The name of the database can also be specified on the command line when staring the GUI as *lstasc lst_project.lstasc*.

2.12. Script Commands

The script commands issued to create the database can be viewed from the View pulldown menu. Use these commands as a template for scripts.

3. EXAMPLE PROBLEMS

The application of the topology code is demonstrated with the help of a few test examples below. The examples are supplied together with the software executables.

3.1. *Fixed Beam with Central Load*

This example is used as a template to demonstrate

1. how to define a problem,
2. how to add a case,
3. how to optimize the topology for a non-extrusion example,
4. analysis of output.

3.1.1. Problem Description

This example simulates a beam that is fixed on both ends. A pole with assigned initial velocity of 10m/s hits the beam in the center. The design part is meshed using 5mm^3 brick elements. The symmetry of the problem is used to design only half-section of the beam. The geometry and loading conditions of the beam are shown in Figure 3-1. The material model used in this example is defined previously.

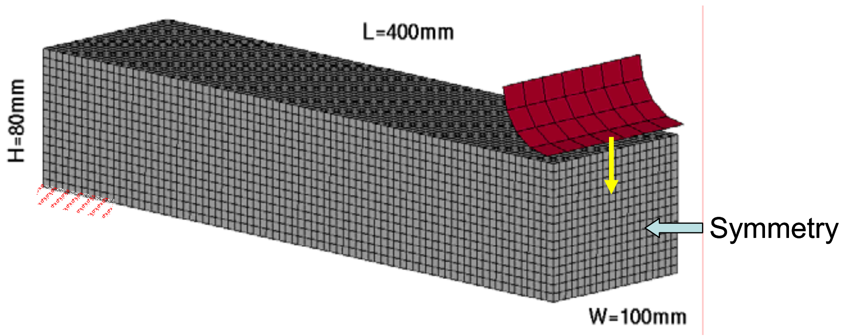


Figure 3-1: Geometry and loading condition of a single-load case example.

3.1.2. Input

The problem has a case named BEAM. The name of the DYNA input deck file is “Beam.dyn”. Part 101 is the design part. A maximum of 100 iterations are used to find the optimal topology. The desired mass fraction is 0.25.

The project input data is saved to the file *lst_project.lstasc* as provided in the examples distribution. Additionally, scripts to recreate the database are also provided. The project database can be investigated using the scripts; use the script in example 5.6.4 to print the project data. The advanced user can conduct the simulations using the LS-DYNA MPP version and hence using a script named “submit_pbs” for the PBS queuing system.

3.1.3. Output

The output of the code is written in the file named lst_output.txt. The error and warning messages are echoed in lst_error and lst_Warning files respectively. The typical output in the lst_output.txt is:

```
ls-dyna analysis time: 161s
it   1:   total IED: 9.933e+03   Mf: 0.250
ls-dyna analysis time: 177s
it   2:  total IED: 9.495e+03   Mf: 0.250  dX: 0.074627 (target: 0.001)
ls-dyna analysis time: 183s
it   3:  total IED: 8.983e+03   Mf: 0.250  dX: 0.077542 (target: 0.001)
ls-dyna analysis time: 187s
it   4:  total IED: 9.252e+03   Mf: 0.250  dX: 0.072176 (target: 0.001)
ls-dyna analysis time: 193s
it   5:  total IED: 9.156e+03   Mf: 0.250  dX: 0.063345 (target: 0.001)
ls-dyna analysis time: 193s
```

a) ***Convergence History***

The convergence is quantified using the change in topology, characterized by the normalized density redistribution, and the total internal energy density as shown in Figure 3-2.

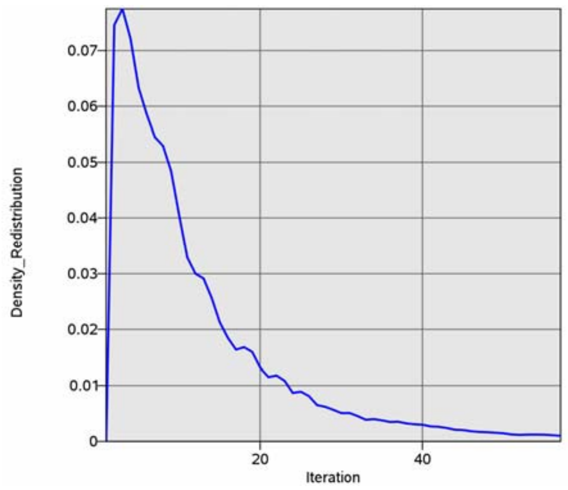


Figure 3-2: Convergence history of the mass redistribution.

The simulation converged after 57 iterations. It was observed that initially there were significant changes in the topology (upto 30 iterations). Afterwards, small changes were made in the topology. There was a drop in the total internal energy density during the early phase of the optimization but it increased during the later iterations. The final topology is visualized in LS-PREPOST.

b) ***Density Contours***

The initial and final topologies are shown in Figure 3-3, and the topologies at different iterations during the evolution process are shown in Figure 3-4.

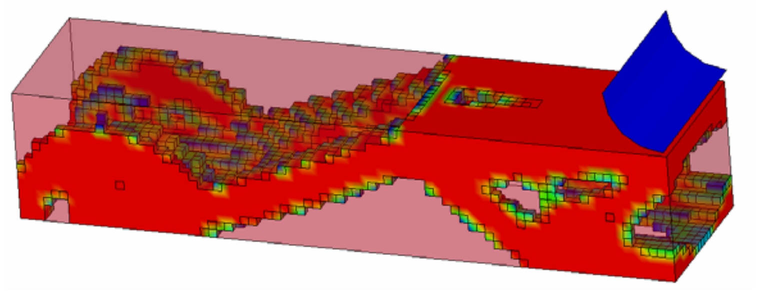


Figure 3-3: Initial and final density contours.

The final topology evolved in a truss-like structure. Many holes were carved to satisfy the mass constraint while reducing the non-uniformity in the distribution of the internal energy density. The final structure was also found to have a reasonably homogenous distribution of the material as was desired.

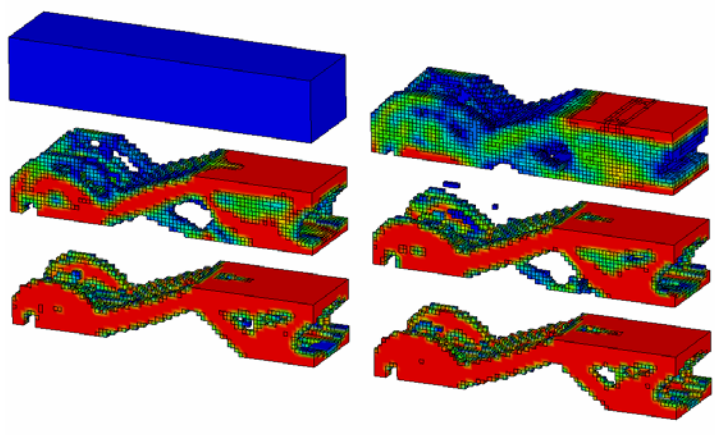


Figure 3-4: Evolution of the geometry shown using density contours.

Topologies at different stages of the evolution process show that the main features of the structure were evolved by iteration 20 (row 2, column 1). Further iterations were necessary to bolster the structure by removing the material from relatively non-contributing zones and redistributing it to the desirable sections such as a 0-1 type topology was evolved.

3.2. *Beam using geometry definitions*

This example demonstrates how to setup a problem with geometry definitions.

The same fixed-beam with a central load example is analyzed with an extrusion and two casting definitions. The symmetry face is also defined as the extruded face. In the input deck file, the elements on the extrusion face were grouped in a solid set (*SET_SOLID). Two different casting conditions were applied in two separate design runs: (i) in the first run casting definition was applied in the Z direction, and (ii) in the second run a two-sided casting definition was applied in the Z direction All other parameters were kept the same.

3.2.1. *Input*

The main differences in this example compared to the non-extrusion example are:

- An extrusion definition is provided.
- A casting definition in Z direction is provided.

The project input data is saved to the file *Extr_Cast.lstasc* and *Extr_Cast2.lstasc* as provided in the examples distribution in the directory *Beam_extr_cast*. Additionally, scripts to recreate the database are also provided. The project database can be investigated using the GUI or a script; use the script in example 5.6.4 to print the project data.

3.2.2. *Output*

a) ***Extrusion and Casting***

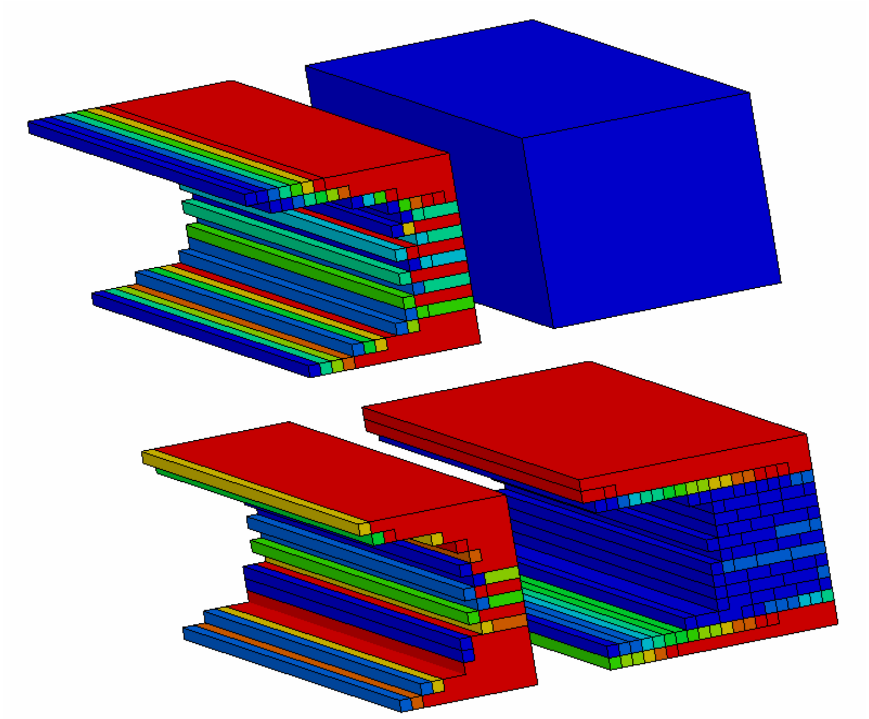


Figure 3-5: Evolution of the beam using extrusion and single-sided casting constraints

Different phases in the evolution are depicted in Figure 3-5. One can see that a lot of material was removed as early. The final geometry evolved by considering the geometry definitions was significantly different than the case when no manufacturing constraints were considered. The C-section evolved makes intuitively sense.

b) ***Extrusion and two-sided casting***

Different phases in the evolution are depicted in Figure 3-5. One can see that a lot of material was removed as early. The final geometry evolved by considering the geometry definitions was significantly different than the case when no manufacturing constraints were considered. The I-section evolved makes intuitively sense.

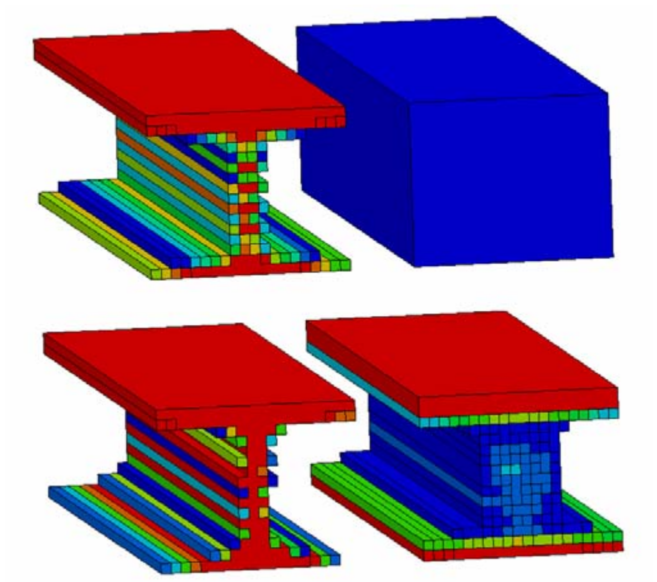


Figure 3-6: Evolution of the beam using extrusion and two-sided casting constraints.

3.3. *Multiple Load Cases*

This example demonstrates a simulation with multiple load cases.

3.3.1. Problem Definition

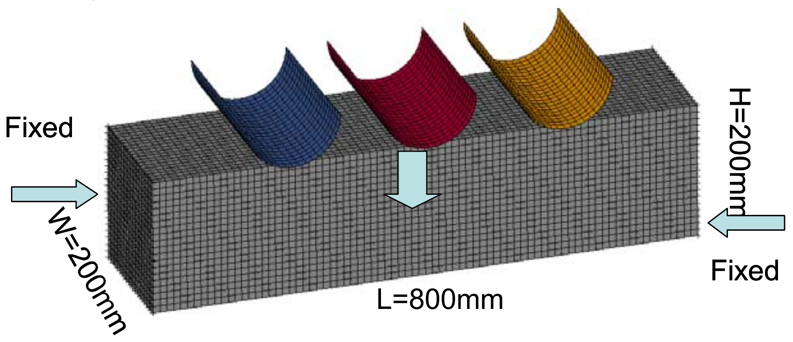


Figure 3-7: The geometry and loading conditions of the multiple load case example.

The geometry and loading conditions for the example are shown in Figure 3-7. This is a fixed-fixed beam with three loads. The design part was meshed with 10mm^3 elements.

3.3.2. Input

The three load cases were identified according to the location of the pole hitting the beam. Side load cases were assigned a unit weight and the center load was assigned a weight of three units. The desired mass fraction for this example was 0.3. A maximum of 100 iterations were allowed. All simulations were run simultaneously.

The project input data is saved to the file *lst_project.lstasc* as provided in the examples distribution. Additionally, scripts to recreate the database are also provided. The project database can be investigated using the scripts; use the script in example 5.6.4 to print the project data. The advanced user can conduct the simulations using the LS-DYNA MPP version and hence using a script named “submit_pbs” for the PBS queuing system.

3.3.3. Output

a) ***Convergence History***

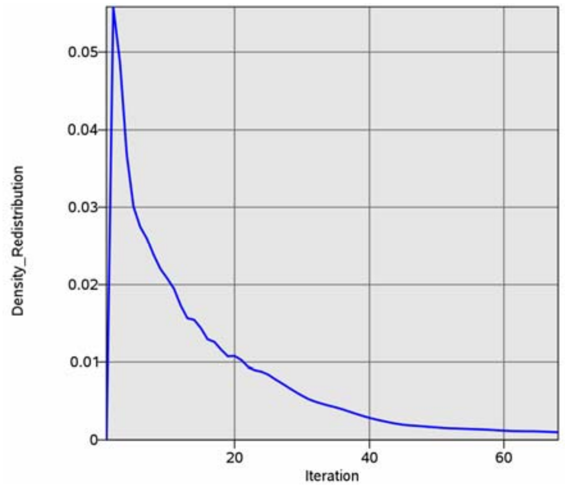


Figure 3-8: Convergence history for multiple-load case example.

The convergence history for the multiple-load example is shown in Figure 3-8. The simulation converged after 67 iterations, though miniscule changes were noted after 40 iterations. As observed before, monotonic reduction in the change in topology was observed. The final structure absorbed approximately 8% less total internal energy.

b) ***Density Contours***

The initial and final structures are shown in Figure 3-9. The final structure evolved in a tabular structure with the two cross-members as legs. The structure had more material in the center section due to the high importance assigned to the center weight. There were many cavities in the structure such that the final structure could be considered equivalent to a truss-like structure as one would expect.

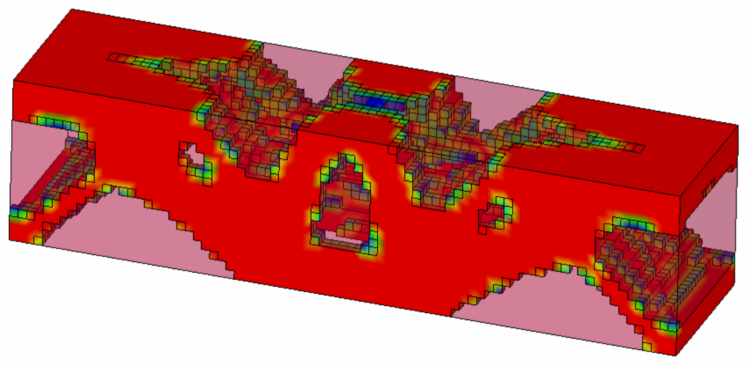


Figure 3-9: Initial and final density contours.

The evolution of the topology under multiple loading conditions is shown in Figure 3-10. While the final form of the structure was largely evolved by 28th iteration (row 2, column 1), the material was re-distributed to remove the low density material and evolve a largely 0-1 (no material or full density material) structure.

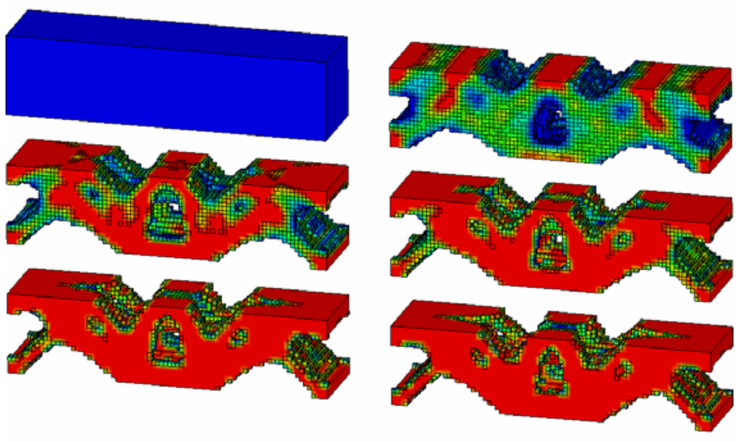


Figure 3-10: Evolution of the geometry for multiple-load case structure.

3.4. Force-Displacement Constraints

The next example demonstrates a simulation with multiple constraints.

3.4.1. Problem Definition

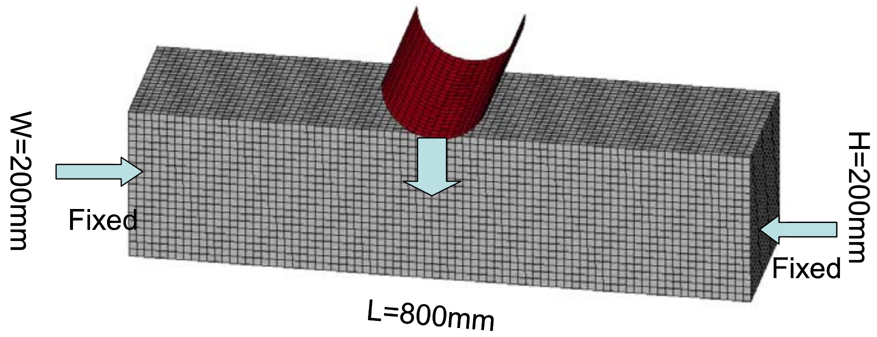


Figure 3-11: The geometry and loading conditions of the multiple constraints example.

The geometry and loading conditions for the example are shown in Figure 3-11. This is a fixed-fixed beam with a central load. The design part was meshed with 10mm^3 elements.

3.4.2. Input

The center load was assigned at the location of the pole hitting the beam. The desired mass fraction for this example was 0.25. A maximum of 100 iterations were allowed. The maximum displacement of the indenter was constrained at 34 units and the maximum y-component of the interface force was limited at 1.45e6 units.

The project input data is saved to the file *lst_project.lstasc* as provided in the examples distribution. Additionally, scripts to recreate the database are also provided. The project database can be investigated using the scripts; use the script in example 5.6.4 to print the project data. The advanced user can conduct the simulations using the LS-DYNA MPP version and hence using a script named “submit_pbs” for the PBS queuing system.

3.4.3. Output

a) ***Convergence History***

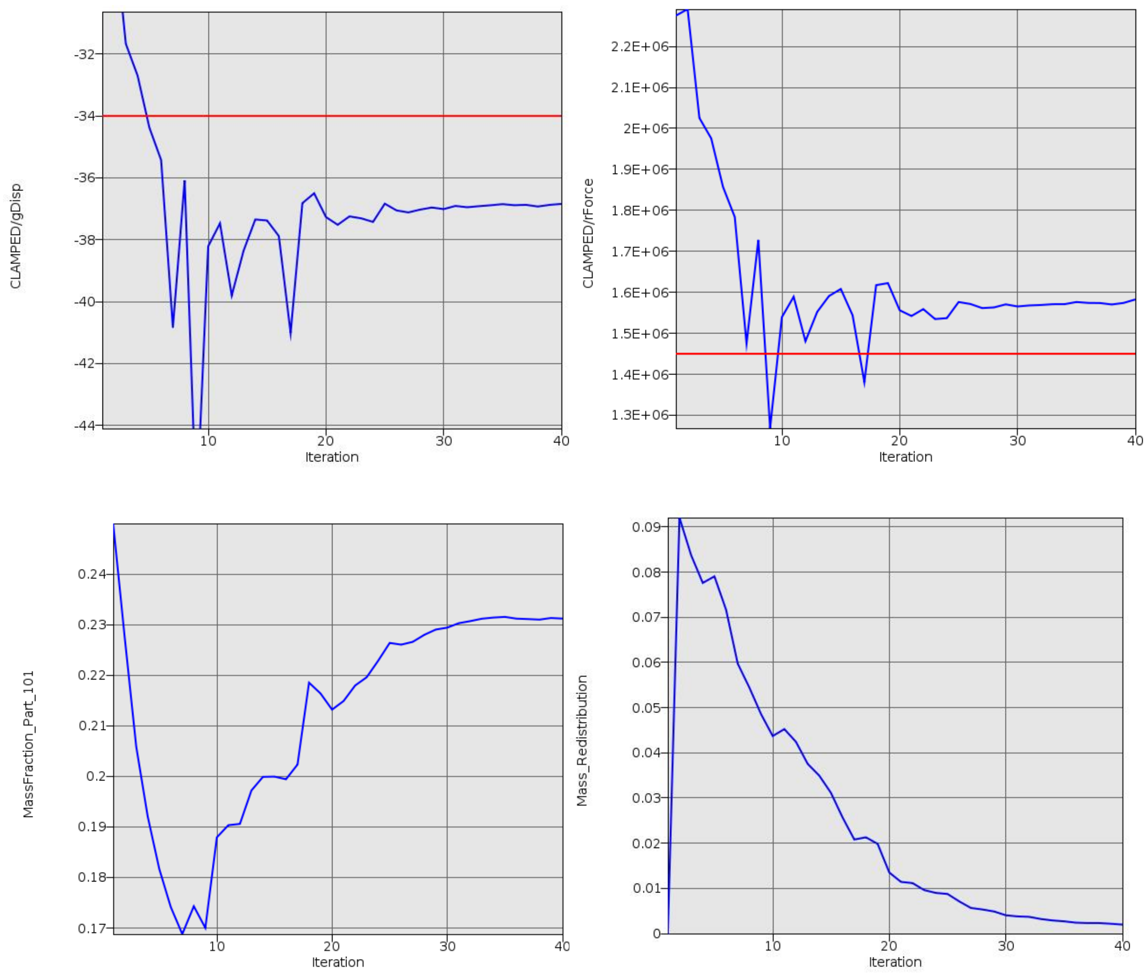


Figure 3-12: Convergence history for the example with multiple constraints.

The convergence history for the multiple-constraints example is shown in Figure 3-12. There were minimal changes in the geometry after 25 iterations and the simulation converged after 40 iterations. While there was largely monotonic reduction in the density redistribution, the constraints and IED were oscillatory in the behavior. The oscillatory behavior of the constraints was due to their conflicting nature where an increase in displacement required an increase in the mass fraction which resulted in higher forces. At optimum, a balance between the two quantities was obtained. It is important to note that the mass fraction for this example was not held constant. Instead, it was automatically adjusted to satisfy the force and displacement constraints though the final mass fraction was fairly close to the desired value.

b) *Density Contours*

The evolution of the topology of the clamped beam with multiple constraints is shown in Figure 3-13. The final structure had many cavities and resembled an optimized truss-like structure. The main cavities in the structure were formulated by the 15th iteration and the structure was fully developed in a largely 0-1 type structure by the 30th iteration. Further redistribution of the material refined this structure between the 30th and the 40th iteration.

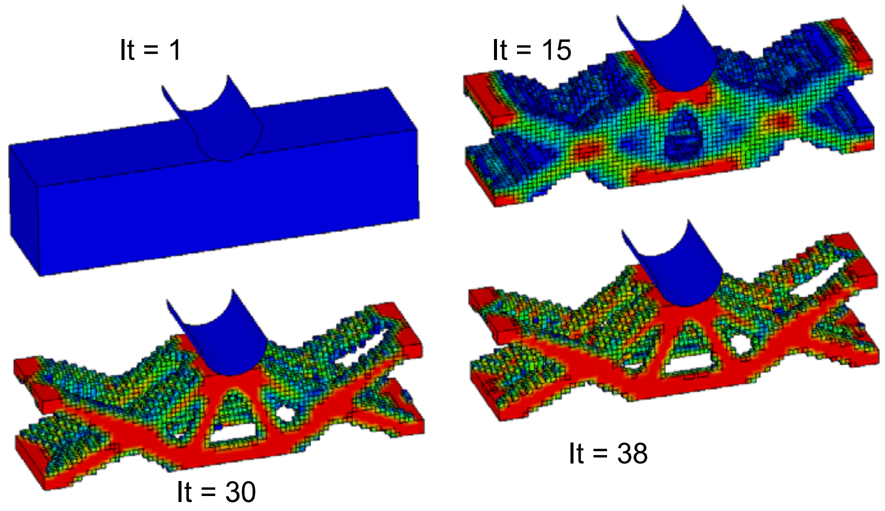


Figure 3-13: Evolution of the geometry for multiple-constrained clamped beam.

3.5. *Linear Static Loading*

The next example demonstrates the topology optimization of a statically loaded structure.

3.5.1. Problem Definition

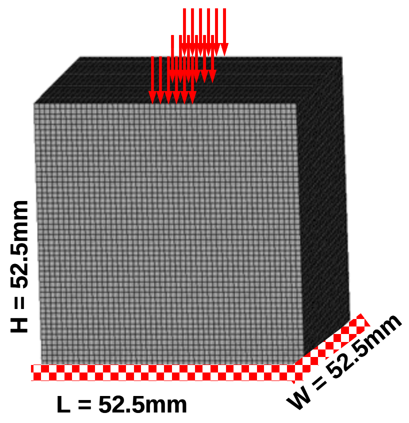


Figure 3-14: The geometry and loading conditions of a statically loaded structure.

The geometry and loading conditions for the example are shown in Figure 3-10. The design part was meshed with 1.05mm^3 elements such that there were approximately 125,000 elements.

3.5.2. Input

In this example, a unit load is applied in the center of the structure. The structure was fixed on the bottom. The problem has a case named *TopLoad*. The simulations are carried out using the double precision SMP version of LS-DYNA (*ls971_double*). The name of the DYNA input deck file is "*LinearStructure.dyn*". Part 102 is the design part. A maximum of 100 iterations are used to find the optimal topology and the desired mass fraction is 0.30.

The project input data is saved to the file *lst_project.lstasc* as provided in the examples distribution. Additionally, scripts to recreate the database are also provided. The project database can be investigated using the scripts; use the script in example 5.6.4 to print the project data.

3.5.3. Output

a) *Convergence History*

The convergence history for the statically loaded structure topology optimization example is shown in Figure 3-15. The simulation converged after 28 iterations, though only minor changes were noted after 20 iterations. As observed before, monotonic reduction in the change in topology was observed. The total internal energy of the structure also decreased with topology evolution.

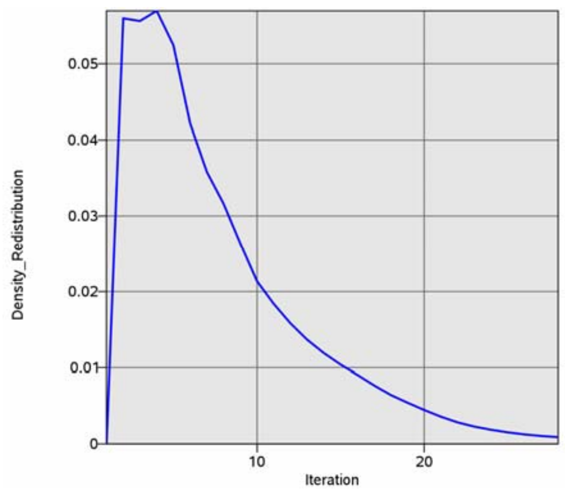


Figure 3-15: Convergence history for linear-static example.

b) *Density Contours*

The initial and final structures are shown in Figure 3-16. The final structure evolved in a column-like structure with wider supports on the faces. The shape of the structure also resembled the best-stress design.

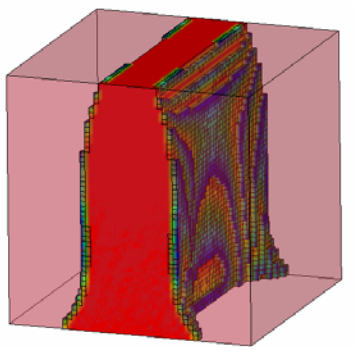


Figure 3-16: Initial and final density contours.

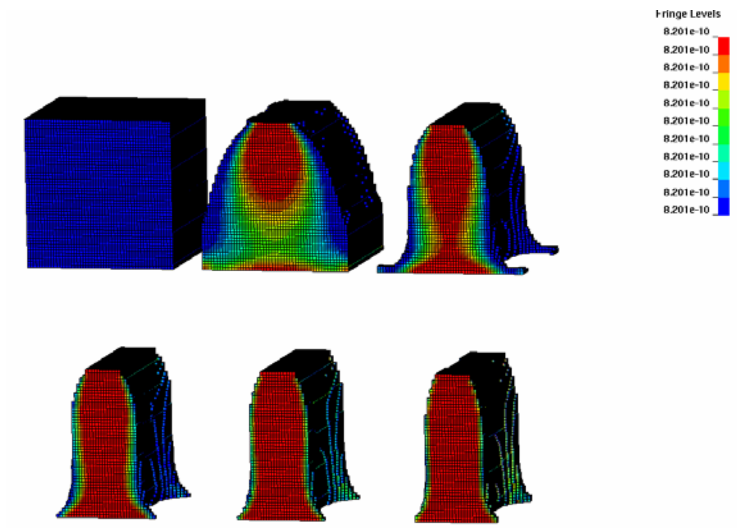


Figure 3-17: Evolution of the geometry for statically loaded structure.

The evolution of the topology under the static loading conditions is shown in Figure 3-17. While the final form of the structure was largely evolved by 17th iteration (first structure in the second row), the material was re-distributed to remove the low-density elements that were not contributing sufficiently to support the load and obtain a homogenous material distribution such that the simulation converged after 28 iterations.

3.6. Shell Example

This example shows how to work with shell structures.

3.6.1. *Problem Definition*

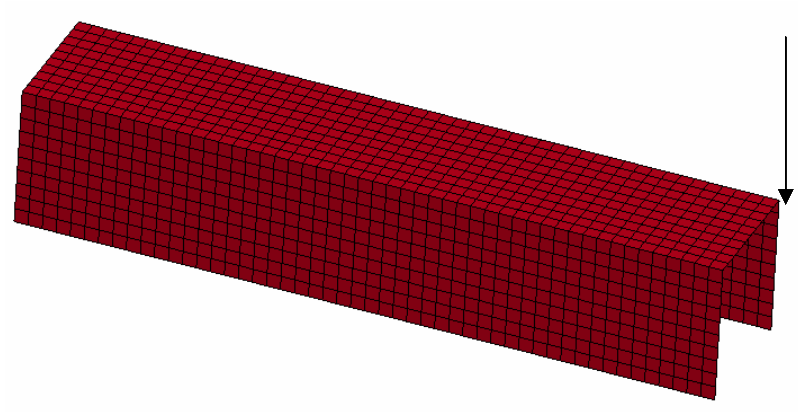


Figure 3-18: The geometry and loading conditions of the shell example. The left side is built-in, while a downward load is applied to the right, back corner.

The geometry and loading conditions for the example are shown in Figure 3-18.

3.6.2. *Input*

The project input data is saved to the file *Shell.lstasc* as provided in the examples distribution. Additionally, scripts to recreate the database are also provided. The project database can be investigated using the scripts; use the script in example 5.6.4 to print the project data.

3.6.3. *Output*

a) ***Convergence History***

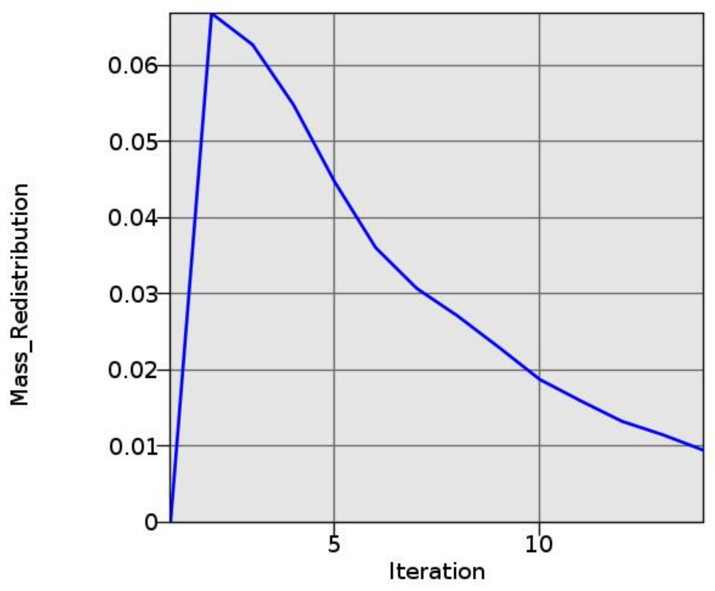


Figure 3-19: Convergence history for the shell example.

The convergence history for the shell example is shown in Figure 3-19. The simulation converged after 14 iterations. There was largely monotonic reduction in the density redistribution.

b) *Final Shell Thicknesses*

The final design is shown in Figure 3-20. The final structure had many cavities and resembled an optimized truss-like structure.

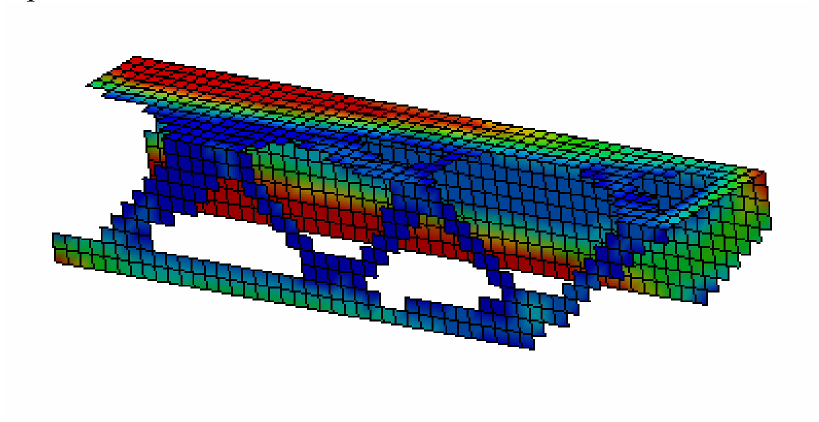


Figure 3-20: Final geometry and thicknesses for the shell problem.

4. TROUBLESHOOTING

This chapter lists some of the most common errors and suggested remedies.

4.1. Executable failing or no output

For the example problems: check that you changed the name of the LS-DYNA executable in the example problem to what is used on your computer.

Provide the complete path for the solver command instead of using *alias*. You may also specify necessary DYNA options in the command, e.g.,

```
/home/Tushar/bin/ls971_single memory=100m
```

4.2. Design Part

The design part is not found: check that the DYNA input deck has the same part id for the design part as specified in the input file. In the case of the multiple load cases, the design domain must remain the same.

4.3. Extrusion Set

The extrusion set is not found: check that the set of elements on the extruded face are grouped under the *SET_SOLID option in the DYNA input deck. The ID of the set is same for all load cases as specified in the input file.

Unable to find all the slaved elements: if the node numbering order is different for some elements are not the same, then the algorithm may fail. Using a different node number will, for example, cause face 1 to be the top face on one element and to be the left face on another element; the algorithm depends on this not happening.

4.4. Negative Volumes

While care has been taken to avoid running into negative volume errors, sometimes the simulation terminates due to negative volume errors.

A user can take several actions to correct this error.
1. Check the CONTACT cards. Note that the failed run probably has elements with soft material interface with elements with harder material; hence care must be exercised in defining master and slave penalty stiffness factors.
2. Specify SOFT=2 option on the control card
3. Increase minimum density fraction (default 0.05 for dynamic problems).

4.5. The LS-DYNA analysis fails if a smaller mass fraction is requested

Possibly the structure is not strong enough to support the load.

Inspect the d3plot results in the failed iteration to understand what happens in the LS-DYNA analysis.

Fixes are to reduce the load, increasing the mass fraction, changing the FE model to be more robust, using a finer mesh, modify your approach keeping in mind that you cannot get a solution from that starting mass fraction, or accepting that a design does not exist at that mass fraction.

4.6. Convergence

For some problems, the code does not converge; instead, oscillations set in. The user must look at the geometry to understand why oscillations are observed. Mostly, oscillations indicate that there is more than one possible optimal solution.

4.7. LS-PREPOST

You may need to install another version of LS-PREPOST into the LS-TaSC installation directory. Please follow the instructions on the LS-PREPOST web site. The name of the executable must be *lsprepost*. Do not use a symbolic link. You may need to investigate the latest version of LS-Prepost 2.4 and 3.1.

4.8. Casting definitions

Using the scripting interface, you can set a debug flag on the *lst_Method* structure, which will dump a definition of the faces to a file for display in LS-PREPOST.

4.9. Mysterious Error when/after calling LS-DYNA and/or Errors involving the LSOPT Environment Variable

Make sure the queuing is set correctly. Specifying the use of a queuing system when none is available may cause (i) mysterious errors or (ii) the LS-DYNA execution not to return after finishing.

Make sure the LSOPT environment variable is not set.

5. APPENDIX A: SCRIPTING

The scripting capability is provided to allow advanced users to customize the application. Normal interaction with the topology optimization code is with the graphical user interface, which issues the scripting commands driving the optimization process.

A script is provided to the program in a file. The commands in a script can perform one of two functions:

- Define the problem and methodology data
- Call the topology design functions

5.1. *The scripting language*

The script commands use the C programming language syntax to manipulate data. Detailed knowledge of the language is not required to use this manual; the example scripts in this manual give enough information. A complete syntax reference is given in the LS-PREPOST customization manual titled "SCRIPTO A new tool to talk with LS-PREPOST" available at http://www2.lstc.com/lspp/index.shtml.

5.2. *Code Execution*

The LS-TaSC code is executed from the command prompt by running the executable (*lstasc_script*). The input command file (script) can be supplied in two manners:

1. With the execution command and a script file name
```
$ lstasc_script lst_script.lss
```
2. The code prompts for the input file, if no input was specified with the execution command
```
$ lstasc_script
Please input command file name:
lst_inp
```

Additionally, you can use the execution command and a database file name

```
$ lstasc_script lst_project.lstasc
```

5.3. *Data-structures*

5.3.1. lst_Root

All input data is encapsulated in a top-level data structure *lst_Root*. The input data is classified in two sub-categories: the problem definition that does not depend on the optimization method, and the optimization method parameters.

```
struct lst_Root {
      struct lst_Problem *Problem;
      struct lst_Method  *Method;
}
```

5.3.2. lst_Method

The parameters used for optimization method are specified in this data-structure.

```
struct lst_Method {
  Int   NumIter;
  Float ConvTol;
  Int   NumDiscreteLevels;
  Int   DumpGeomDef;
  Int   StoreFieldHist;

}
```

`NumIter:` The maximum number of iterations allowed is specified.

`ConvTol:` The convergence tolerance is the termination criterion used to stop the search when the topology has evolved sufficiently. If $ConvTol \leq 0.0$, then this input would be ignored, and the default will be used.

`NumDiscreteLevels:` Resolution or the number of steps in the gradation of the material of the part being design. The default value should suffice for almost all problems.

`DumpGeomDef:` Set this to a non-zero value to obtain debugging information for casting constraints. Files will be created which can be viewed in LS-PREPOST showing the master face (free) elements, and the elements chained to the master elements.

`StoreFieldHist:` Set this to a non-zero value to obtain the IED histories in the View panel.

5.3.3. lst_Problem

The details of the problem is given in this data structure. The definition is as follows:

```
struct lst_Problem {
  struct lst_Case *CaseList;
  struct lst_Part * PartList;
  Char * Description;
}
```

`CaseList:` The user provides the details of the simulation in this data structure. As the name suggests, the `CaseList` is the list of all load cases. For multiple load cases, the user would specify one *case* per load case. A complete description is given in a following section.

`PartList:` The user provides the details of the parts in this data structure. As the name suggests, the `PartList` is the list of all parts. A complete description is given in the next section.

`Description:` This optional string is used to describe the problem.

5.3.4. lst_Part

The details of a part are:

```
     struct lst_Part { Int ID;
                       Int Continuum;
                       Float MassFraction;
                       Float ProxTol;
                       Float MinVarValue;
                       struct lst_Geometry * GeometryList;
                       struct lst_Part * Next; }
```

`ID:` Each part is identified with a unique id as in the LS-DYNA input deck.
The design domain for topology optimization is identified as all of the parts given.

`ProxTol:` All elements within a radius of proximity tolerance would be considered as the neighbors of an element.

`MinVarValue:` Elements with a density of less than this will be deleted.

`MassFractionBound:` The material constraint for the topology optimization is necessary for the optimization. An appropriate value ($0.05 < x < 0.95$) is supplied here.

`Continuum:` Whether the part is a solid or a shell. Solids have a value of 1, while shells have a value of 2.

`GeometryList:` These are the geometry and manufacturing constraint on a part. A complete description is given in the next section.

`Next:` The next part in this linked list. A value of NULL indicates that this is the final part.

5.3.5. lst_Geometry

The details of a geometry definition are:

```
     struct lst_Geometry { Char *Name;
                           Int Type;
                           Int CID;
                           Int Set;
                           Int ExtructionDir;
                           Int MirrorPlane;
                           struct lst_Geometry * Next; };
```

`Name:` Each geometry definition is identified with a unique name. The name is used to identify the geometry constraint in the output.

`Type:` The type of extrusion. 2 is an extrusion, 3 is a symmetry constraint, 4 is a single sided casting constraint, and 5 is a double sided casting constraint.

`Set:` To design an extruded part, the user firstly creates a set of all solid elements that would be extruded (SET_SOLID). The *id* of this set is specified in the input deck to identify the extrusion set.

`ExtrusionDir:` X=1 Y=2 Z=3

`MirrorPlane:` The mirror plane for a symmetry constraint XY=1 YZ =2 ZX = 3.

`Next:` The next geometry definition in this linked list. A value of NULL indicates that this is the final geometry definition.

5.3.6. lst_Case

The details of the simulation setup are given in this data structure.

```
     struct lst_Case {
       Char                    *Name;
       Char                    *SolverCommand;
       Char                    *InputFile;
       Int                     AnalysisType;
       Float            Weight;
       struct lst_Constraints *ConstraintList;
       struct lst_JobInfo     *JobInfo;
       struct lst_Case        *Next;
     }
```

`Name:` Each case is identified with a unique name e.g., TRUCK. The same name would be used to create a directory to store all simulation data.

`SolverCommand:` The complete solver command or script (e.g., complete path of LS-DYNA executable) is specified.

`InputFile:` The LS-DYNA input deck path is provided.

`AnalysisType:` The topology optimization code can be used to solve both static and dynamic problems. The user identifies the correct problem type by specifying the correct option:

Type	Option
STATIC	1
DYNAMIC	2

`Weight:` The weight associated with a case is defined here. This enables the user to specify non-uniform importance while running multiple cases.

`ConstraintList:` This data structure holds the information about different constraints associated with this case. See the following section for more details.

`JobInfo:` The user specifies details of the queuing system and number of simultaneous processes in this data structure.
`Next:` The next case in this linked list. A value of NULL indicates that this is the final geometry case.

Note that the word `case` is a reserved word in the C programming language.

5.3.7. lst_Constraint

The structural constraints for a discipline are specified in the following data structure:

```
struct lst_Constraint {
    char * Name;
    Float UpperBound;
    Float LowerBound;
    Char * Command;
    struct lst_Constraint *Next;
}
```

`Name:` The name of each constraint is a unique character identifier.

`UpperBound/LowerBound:` The upper and lower bounds on a constraint are specified using these variables. If there is no upper bound, a value of 1.0e+30 must be specified for `UpperBound`. Similarly, a value of -1.0e+30 should be used for `LowerBound` when there is no lower bound.

`Command:` The definition of each constraint provides interface to LS-DYNA® databases. The data extraction from both *binout* and *d3plot* databases are supported.

5.3.8. lst_JobInfo

This data structure contains the LS-DYNA® job distribution information. Create and set this data structure to change the default of running LS-DYNA® locally as a single process.

```
struct lst_JobInfo {
    Int NumProc;
    Int Queuer;
    Char ** EnvVarList;
}
```

`NumProc:` This parameter indicates the number of processes to be run simultaneously. A value of zero indicates all processes would be run simultaneously.

`Queuer:` This parameter is used to indicate the queuing system. Different options are tabulated below.

Q-system	Option	Q-system	Option	Q-system	Option
QUEUE_NIL	0	NQS	4	BLACKBOX	8
LSF	1	USER	5	MSCCP	9
LOADLEVELER	2	AQS	6	PBSPRO	10
PBS	3	SLURM	7	HONDA	11

By default, no queuing system would be used.

`EnvVarList`: These parameters are passed to the remote machine by the queuing system. The `lst_JobInfoAddEnvVar` command is used to set the values.

5.4. *Interactions with the Data Structures*

To specify the input data, the user needs to communicate with the program data structures. These data structures are accessed by the user *via* a script that follows the syntax of C programming language. So the user needs to first define the data structure and then populate the input data.

5.4.1. Definition

Each script must include the following command to access necessary data-structures.

```
lst_Root *root = lst_RootGet();
```

The root data structure encapsulates both problem and method data and therefore always needs to be accessed.

5.4.2. Initialization

During initialization, the user provides the necessary input data.

a) ***Adding Case Data***

The solver information is added to the problem data using the `lst_ProblemAddCase` function, defined as follows:

```
lst_ProblemAddCase( lst_Problem, Char *CaseName, Char
*SolverCmd, Char * InputFileName", Int analysisType,
Float Weight );
```

The last two arguments *analysisType*, and *weight* are optional. If not specified then the program will determine whether it is a non-linear analysis and set the weight to 1.0.

Example: Add two load cases

1. This load case uses a queuing system for a nonlinear structural problem

```
lst_ProblemAddCase( root->Problem, "LEFT_LOAD",
"submit_pbs", "MyInputL.k", 2, 0.5);
```

2. Second load case uses a standalone DYNA program for a linear structural problem

```
lst_ProblemAddCase( root->Problem,
"RIGHT_LOAD","ls971_single", "MyInputR.k", 1, 0.9);
```

b) ***Accessing a Specific Case Structure***

The cases are stored in a linked list in the `lst_Problem` structure. Also a pointer to the `lst_Case` structure is returned when it is created. Note that the word `case` is a reserved word in the C programming language.

```
lst_Case * cse1 = root->Problem->CaseList;
lst_Case * cse2 = root->Problem->CaseList->Next;
```

```
     lst_Case * cse4 = cse1->Next->Next->Next;
     lst_Case * cse  = lst_ProblemAddCase( root->Problem,
“RIGHT_LOAD”,"ls971_single", "MyInputR.k", 2, 1);
```

c) ***Adding Constraints***

A user can add constraints to each case using the following command:

```
     lst_CaseAddConstraint ( struct lst_Case* cse, Char *
     constraintName, Float UpperBound, Float LowerBound,
     Char *constraintCommand );
```

Example: Adding two constraints to a case

1. Adding a displacement constraint:
   Maximum resultant displacement of part defined by id=101 should be less than 7.25 units

```
lst_CaseAddConstraint (root->Problem->CaseList, “gDisp”,
7.25, -1.0e+30, “D3PlotResponse -pids 101 -res_type ndv -
cmp result_displacement -select MAX -start_time 0.00”);
```

2. Adding a force constraint:
   Maximum y-force on the master side of the interface defined by id=9 should be smaller than 2.0e5 units.

```
lst_CaseAddConstraint (root->Problem->CaseList, “rForce”,
2.0e5, -1.0e+30, “BinoutResponse -res_type RCForc -cmp
y_force -id 9 -side MASTER -select MAX -start_time
0.00”);
```

It is recommended to obtain the command definition using the GUI. The LS-OPT manual can also be consulted on how to create the string.

d) ***Adding Part Data***

A user can add parts to the problem using the following command:

```
     struct lst_Part * lst_ProblemAddPart( struct
     lst_Problem *prob, Int partId, Float massFracB, Double
     minx, Double proxTol );
```

with the items in the command as explained for the part structure. The last two arguments (the minimum variable value and the neighbor radius) are optional.

Example: Adding a part

```
struct lst_Part * prt = lst_ProblemAddPart( root-
>Problem, 102, 0.3 );
```

e) *Accessing a Part*

The parts are stored in a linked list in the `lst_Problem` structure. In addition, a pointer to the `lst_Part` structure is returned when it is created.

```
     lst_Part * part1 = root->Problem->PartList;
     lst_Part * part2 = root->Problem->PartList->Next;
     lst_Part * part4 = part1->Next->Next->Next;
     lst_Part * prt  = lst_ProblemAddPart( root->Problem,
101, 0.3 );
```

f) *Adding Geometry Data*

A user can add geometry constraints to the problem using the following commands:

```
     struct lst_Geometry * lst_PartAddGeometryExtrusion(
     struct lst_Part *, const char * name, long set, long
     CID, long dir );
     struct lst_Geometry * lst_PartAddGeometryExtrusionConn(
     struct lst_Part *, const char * name, long set );
     struct lst_Geometry * lst_PartAddGeometrySymmetryXY(
     struct lst_Part *, const char * name, long CID );
     struct lst_Geometry * lst_PartAddGeometrySymmetryYZ(
     struct lst_Part *, const char * name, long CID );
     struct lst_Geometry * lst_PartAddGeometrySymmetryZX(
     struct lst_Part *, const char * name, long CID );
     struct lst_Geometry * lst_PartAddGeometry1SideCasting(
     struct lst_Part *, const char * name, long dir, long
     CID );
     struct lst_Geometry * lst_PartAddGeometry2SideCasting(
     struct lst_Part *, const char * name, long dir, long
     CID );
```

g) *Adding Job Distribution Data*

Details about running the simulation job for each case can be added by creating a `JobInfo` structure and using `lst_CaseSetJobInfo` function. The syntax is as follows.

```
     lst_JobInfo * ji = lst_JobInfoNew();
     ji->NumProc = 1;
     ji->Queuer = 3;
     lst_CaseSetJobInfo( aCase, ji );
```

For jobs submitted using a queuing system, the values of the environment variables can be set on the remote system, if required, using the `lst_JobInfoAddEnvVar` command. The command has the following syntax:

```
     lst_JobInfoAddEnvVar( struct JobInfo* ji, char *
     variableName, char * value );
     lst_JobInfoDeleteEnvVar( struct JobInfo* ji, char *
     variableName );
```

Example: Adding simulation information to the two cases

1. Adding JobInfo to the case LEFT_LOAD that uses PBS queuing system,

```
     lst_JobInfo * ji = lst_JobInfoNew();
     ji->NumProc = 0;
     ji->Queuer = 3;
     lst_JobInfoAddEnvVar( ji, “LS_NUM_ABC”, “5”);
     lst_CaseSetJobInfo( left_load_case, ji );
```

2. Adding JobInfo to the case RIGHT_LOAD that does not use any queuing system,

```
     lst_JobInfo * ji = lst_JobInfoNew();
     ji->NumProc = 1;
     ji->Queuer = 0;
     lst_CaseSetJobInfo( right_load_case, ji );
```

h) ***Specifying Optimization Method Parameters***

Once the root data structure is obtained, the data in Method data structure can be directly manipulated.

1. Specify the maximum number of iterations

```
     root->Method->NumIter  = Int;
```

2. Provide convergence tolerance

```
     root->Method->ConvTol  = Float;
```

3. To specify proximity tolerance use

```
     root->Method->ProxTol  = Float;
```

5.4.3. Execution Functions

a) ***Saving the Project Data***

The program save the project input data in form of a XML database.

```
     lst_RootWriteDb( root );
```

A default filename of “lst_project.lstasc” is used, but you may specify the filename.

```
     lst_RootWriteDb(root, "filename.xml" );
```

b) ***Reading the Project Data***

The project input data can be read from disk as:

```
     lst_Root *root = lst_RootReadDb();
```

A default filename of “lst_project.lstasc” is used, but you may also specify the filename.

```
     lst_Root *root = lst_RootReadDb( “filename.xml” );
```

c) *Create Topology*

Following command computes the topology:

```
    lst_CreateTopology(root);
```

The status of each simulation can optionally be reported every "Interval" seconds as shown in the following command:

```
    lst_CreateTopology(root, Interval);
```

d) *Cleaning the directory*

The files created in the directory can be removed:

```
    lst_CleanDir("databaseFileName.lstasc");
```

The filename was specified in this case; if omitted, the default of "lst_project.lstasc" will be used. All of the files created for the analysis, except the database, will be removed.

5.5. *Accessing Results*

These commands access the LS-TaSC database and the LS-DYNA® binout database using the LSDA (LSTC Data Archival) interface. Read this section together with the LSDA documentation available from the LSTC ftp site.

Open a database

Command	Int handle lsda_open(Char *filename)
Example	Int fout = lsda_open("lstasc.lsda");
handle	An Int used to indentify this file in further actions.
filename	A string giving the filename or path to the database.

Close a database

Command	Int success lsda_close(Int handle)
Example	Int flag = lsda_close(fout);
Success	An Int specify whether the command succeeded (>0).
handle	An Int identifying the lsda database.

Change to a database directory

Command	Int success lsda_cd(Int handle, Char * dirName)
Example	Int flag = lsda_cd(fout, "Design#4");
Success	An Int specify whether the command succeeded (>0).
handle	An Int identifying the lsda database.
dirName	A String specifying the database directory.

Get the current directory in a database

Command	Char *dirName lsda_getpwd(Int handle)
Example	Char *currDir = lsda_getpwd(fout);
dirName	A String with the name of the current directory in the database. Do not free this

	string.
handle	An Int identifying the lsda database.

Print the content of the current directory

Command	Int numItems lsda_ls(Int handle)
Example	Int n = lsda_ls(fout);
numItems	An Int specifying the number of items (directories and data vectors) in this directory.
handle	An Int identifying the lsda database.

Get Integer data

Command	Int * data lsda_getI4data(Int handle, Char * variableName, Int * numValues)
Example	Int * results = lsda_getI4data(fout, “elementLabels”, &numV);
data	A pointer to Int containing the data. You must free this pointer after using.
handle	An Int identifying the lsda database.
variableName	The name of the results.
numValues	The length of the data vector (the number of items).

Get Float data

Command	Float * data lsda_getR4data(Int handle, Char * variableName, Int * numValues)
Example	Float * results = lsda_getR4data(fout, “xx-stress”, &numV);
data	A pointer to Float containing the data. You must free this pointer after using.
handle	An Int identifying the lsda database.
variableName	The name of the results.
numValues	The length of the data vector (the number of items).

5.6. *Example Script*

5.6.1. Retrieving a value from the project database

Retrieving a value from the database is simple: opened the project database and the value is available.

```
lst_Root *root = lst_RootReadDb();
print( "Existing number of iterations: ", root->Method->NumIter );
```

5.6.2. Restart for an additional iteration

Requiring four lines of code, this is slightly more complex than the previous example.

```
lst_Root *root = lst_RootReadDb();
root->Method->NumIter = root->Method->NumIter + 1;
lst_RootWriteDb( root );
lst_CreateTopology(root);
```

5.6.3. *Creating a topology database*

An example script is shown here. The example performs topology optimization of a single load case problem using extrusion mode.

```
lst_Root *root = lst_RootGet();

lst_Case   *cse   =   lst_ProblemAddCase(   root->Problem,   "TOPLOAD",
"/data1/tushar/submit_pbs", "small_example.k", 2, 1 );

lst_JobInfo *ji = lst_JobInfoNew();
ji->NumProc = 0;
ji->Queuer = 3;
lst_CaseSetJobInfo( cse, ji );

lst_Part *prt = lst_ProblemAddPart( root->Problem, 103, 0.3 );
lst_PartAddGeometryExtrusionConn( prt, "Extr", 1 );

lst_RootWriteDb( root );
```

5.6.4. *Printing the content of the project database*

The following script prints the content of a project XML database.

```
define:
int Print_JobInfo( lst_JobInfo *jInfo, char * whitespace )
{
int i;

 print( whitespace, "*** JobInfo ***\n" );
 print( whitespace, "\tNumProc\t\t", jInfo->NumProc, "\n" );
 print( whitespace, "\tQueuer\t\t", jInfo->Queuer, "\n" );
 if( jInfo->EnvVarList ) {
     i = 0;
     while( jInfo->EnvVarList[i] ) {
         print(  whitespace,  "\tEnvVar\t\t",  jInfo->EnvVarList[i],  "\n"
);
         i = i+1;
     }
 }
}

define:
int Print_Case( lst_Case *cse, char * whitespace )
{
struct lst_JobInfo *jInf;
    print( whitespace, "*** Case ***\n" );
    print( whitespace, "\tName\t\t\"", cse->Name, "\"\n" );
    print( whitespace, "\tSolverCommand\t\"", cse->SolverCommand, "\"\n"
);
    print( whitespace, "\tInputFile\t\"", cse->InputFile, "\"\n" );
    print( whitespace, "\tWeight\t\t", cse->Weight, "\n" );
    print( whitespace, "\tAnalysisType\t", cse->AnalysisType, "\n" );

    jInf = cse->JobInfo;
    Print_JobInfo( jInf, "\t\t" );
}

define:
int Print_Geom( lst_Geometry *geom, char * whitespace )
{
    print( whitespace, "*** Geometry ***\n" );
    print( whitespace, "\tName \t\t", geom->Name, "\n" );
    print( whitespace, "\tType \t\t", geom->Type, "\n" );
    print( whitespace, "\tCID \t\t", geom->CID, "\n" );
    print( whitespace, "\tExtructionDir\t\t", geom->ExtructionDir, "\n"
);
    print( whitespace, "\tMirrorPlane\t\t", geom->MirrorPlane, "\n" );
}

define:
int Print_Part( lst_Part *prt, char * whitespace )
{
struct lst_Geometry *geom;

    print( whitespace, "*** Part ***\n" );
    print( whitespace, "\tID \t\t", prt->ID, "\n" );
    print( whitespace, "\tMassFraction\t", prt->MassFraction, "\n" );
    print( whitespace, "\tMinVarValue\t", prt->MinVarValue, "\n" );
    print( whitespace, "\tProxTol\t\t", prt->ProxTol, "\n" );

    geom = prt->GeometryList;
    while( geom ) {
        Print_Geom( geom, "\t\t" );
        geom = geom->Next;
    }
}

define:
int Print_Problem( lst_Problem *prob, char * whitespace )
{
struct lst_Case *cse;
struct lst_Part *prt;

    print( whitespace, "*** Problem ***\n" );
    print( whitespace, "\tDescription\t\t", prob->Description, "\n" );
    print( whitespace, "\tNumCase\t\t\t", prob->NumCase, "\n" );
    print( whitespace, "\tNumPart\t\t\t", prob->NumPart, "\n" );

    cse = prob->CaseList;
    while( cse ) {
        Print_Case( cse, "\t" );
        cse = cse->Next;
    }

    prt = prob->PartList;
    while( prt ) {
        Print_Part( prt, "\t" );
        prt = prt->Next;
    }
}

define:
int Print_Method( lst_Method *meth, char * whitespace )
{
    print( whitespace, "*** Method ***\n" );
    print( whitespace, "\tNumIter\t\t\t", meth->NumIter, "\n" );
    print( whitespace, "\tConvTol\t\t\t", meth->ConvTol, "\n" );
    print( whitespace, "\tNumDiscreteLevels\t", meth->NumDiscreteLevels,
"\n" );
    print( whitespace, "\tDebugGeomDef\t\t", meth->DebugGeomDef, "\n" );
}

/**********************    PROGRAM    TO    PRINT    LST    DATABASE
*********************/
```

```
struct lst_Root *root;
struct lst_Problem *prob;
struct lst_Method *meth;

root = lst_RootReadDb( );

prob = root->Problem;
Print_Problem( prob, "" );

meth = root->Method;
Print_Method( meth, "" );
```

5.6.5. Printing the content of the results database

```
Int flag, numV, iter = 1;
Char dirName[1024];
Float *data, *wmEID, *aveChng;

print( "\"ItNum\",\"Total_IED\",\"Density_Redistribution\"\n" );
Int handle = lsda_open( "lst.binout" );
lsda_cd( handle, "/" );
sprintf( dirName, "/design#%d", iter );
while ( lsda_cd( handle, dirName ) == 1 ) {
  wmEID = lsda_getR8data( handle, "Total_IED", &numV );
  aveChng = lsda_getR8data( handle, "Density_Redistribution", &numV );

  print( iter, ", ", wmEID[0], ", ",  aveChng[0], "\n");

  iter = iter+1;
  sprintf( dirName, "/design#%d", iter );

  free( wmEID ); free( aveChng );
}

lsda_close( handle );
```

6. APPENDIX B: THEORY

I must say it looks a bit like science fiction to many people. – Ofir Shor, June 2009, while evaluating the alpha version.

6.1. *Background*

The traditional approach for solving topology optimization problems is based on sensitivity analysis that is inexpensive to obtain for linear-static problems. However, deriving analytical sensitivities for dynamic analysis is very difficult due to the complex interactions among material nonlinearities, geometry and mesh, and transient nature of load and boundary conditions. Numerical computation of sensitivities is also not practical due to the high computational expense. Hence the conventional sensitivity based approach of topology optimization is not practical for crashworthiness problems. To overcome the aforementioned difficulties in topology optimization, an optimality criteria approach was proposed. This approach does not require gradients and hence there is no need to compute the sensitivities. In previous versions, the approach was refer to as Hybrid Cellular Algorithm [1,2], but we found older views of the technology to be more representative of what is currently actually implemented.

6.2. *Implementation*

The algorithm for structural optimization is shown pictorially in Figure 6-1. After defining the problem, the topology is evolved using the simple rules defined on the variables. The constraints are accommodated during the state update procedure.

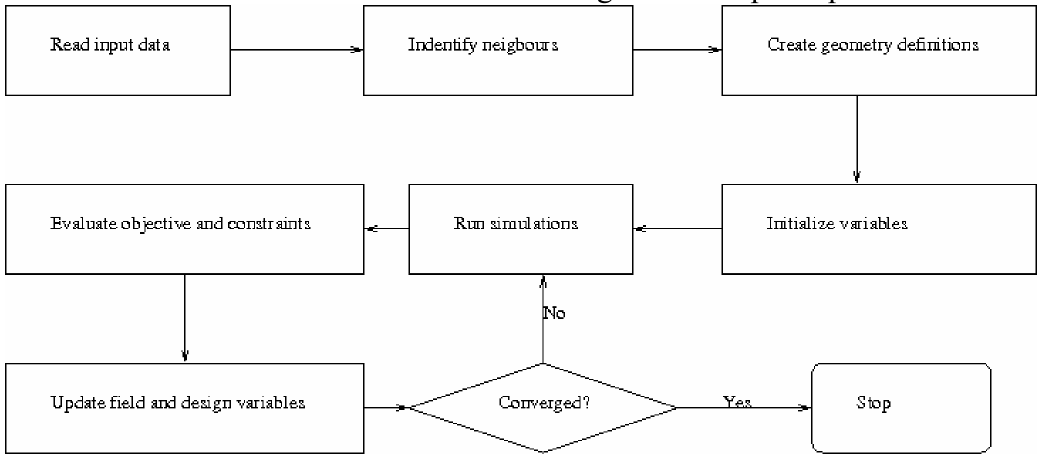

Read input data
Indentify neigbours
Create geometry definitions
Initialize variables
Run simulations
Evaluate objective and constraints
Update field and design variables
Converged?
No
Yes
Stop

Figure 6-1: The topology optimization algorithm

6.2.1. Definition

The input data is used to identify the design domain and design material model. The input data comprises of method data e.g., number of iterations, convergence tolerance, and the problem data, e.g. load cases, design part, etc.

6.2.2. Creating the variables

The finite element model is mapped to design variables. Each design variables is assigned to a solid element in the design domain. For extrusion and symmetry constraints, the equality constraints are defined between the variables. For casting constraints, inequality constraints are established.

6.2.3. Filtering of results

Past work were based on the structured grid arrangement of cells. This assumption would breakdown for industrial applications where structured grids are not always possible. Hence, a radius based strategy is used to identify neighbors. In this strategy, a virtual sphere of user-defined radius is placed at the centroids of an element. All elements that are within this sphere are considered the neighbors of the corresponding element, and the results are averaged over the elements in the neighborhood

$$U_i = \sum_{j=1}^{n} U_j \Big/ \sum_{j=1}^{n} 1. \tag{2}$$

6.2.4. Material Parameterization

The material model is parameterized using a so-called *density approach*. In this approach, a design variable is directly linked to the individual material element such that each variable has its own material model. The material properties corresponding to the values of design variables are obtained using an appropriate interpolation model. The solid isotropic material with penalization (SIMP) model [6] is the most popular interpolation method. This model is power law approach that drives the intermediate material properties towards the boundaries to obtain a 0-1 topology. According to SIMP model, the material properties are defined as,

$$\rho(x) = x\rho_0, \tag{3}$$

$$E(x) = x^p E_0, \tag{4}$$

$$\sigma(x) = x^q \sigma_0, \tag{5}$$

$$E_h(x) = x^q E_{h0}, \tag{6}$$

where ρ denotes the density of the material, E represents the Young's modulus, σ is the yield stress, and E_h is the strain hardening modulus. The last two material properties represent material non-linearities and are required for dynamic problems like crash that involve material yielding. The subscript '0' refers to the base material properties. The design variable *x,* also known as relative density, varies from 0 to 1 where '0' indicates void and '1' represents full material. A more detailed description of the material model parameterization, one should refer to Bendsøe and Sigmund [7], and Patel [8]. The elements with design variable value less than a user-defined minimum value are deleted to improve numerical stability.

6.2.5. *Design Objectives and Constraints*

The typical goal of topology optimization is to obtain maximum utility of the material. Compliance and the strain energy density are the most commonly used objectives for linear-static problems. For dynamic problems like crashworthiness simulations, the structure needs to absorb maximum energy while maintaining the structural integrity and keeping the peak loads transmitted to the occupants low. Following the formulation proposed by Patel [8], the goal of obtaining uniform internal energy density in the structure is defined as the objective for optimization. This concept is similar to the fully-stressed design and uniform strain energy density approaches (Haftka and Gurdal [9], Patnaik and Hopkins [10]) that are well established in literature for linear-static problems.

The optimization problem is formulated as,

$$\min_{x} \sum_{i=}^{N} \sum_{j=1}^{L} \left(w_j U_j(x_i) - U_j^* \right), \tag{7}$$

$$\begin{aligned} subject\ to: \ & \sum_{i=1}^{N} \rho(x_i) V_i \le M^* \\ & C_j^l \le C_j \le C_j^u, \ j = 1,2,...,J \\ & x_{\min} \le x_i \le 1.0. \end{aligned} \tag{8}$$

where U represents the internal energy density of the i^{th} element, V_i is the volume of i^{th} element, U^* represents internal energy density set point, and C_j is the j^{th} constraint. There are L load cases with a total of J constraints. The superscripts '*l*' and '*u*' represent lower and upper bounds on the constraints, respectively.

6.2.6. *Design Variable Initialization*

The design variables are initialized to satisfy the material constraint. All elements are assigned the same design variable values. All associated field variables are also initialized to zero.

6.2.7. *Simulation to Obtain Field Variables*

The elements in the finite element model are modified by changing the material models, adding or deleting elements, at each iteration. So the input deck is re-written at each iteration. This modified input deck is analyzed using LS-DYNA® [11]. One can take advantage of multiple processors using the MPP version of LS-DYNA. The relevant field variables for all elements are obtained from the output to completely define the state of each variable. For multiple load case conditions, the state variable is based on the output from simulations of different load cases.

For dynamic problems, it was observed that accounting for the history of evolution induces stability by reducing the element deletion rate. Hence, the field variable (internal energy density) of i^{th} variable at iteration t is updated by defining a weighted sum on the field variable of three previous iterations as follows,

$$U_i^t = \sum_{j=0}^{3} (x_i)^{j+1} U_i^{t-j} \Big/ \sum_{j=0}^{3} (x_i)^{j+1}. \tag{9}$$

where x_i is the design variable associated with the i^{th} variable at iteration t.

6.2.8. Constraint Handling

In presence of constraints other than the mass constraints, the target mass constraint is adjusted to satisfy the structural constraints. The mass target (M^*) is increased in proportion to the constraint violation for all constraints except force constraints for which the mass target is reduced.

$$\begin{aligned} M^* &= M^* + \Delta M, \\ \Delta M &= \left(\sum_j K_j^c \varepsilon_j \right) / J, \end{aligned} \tag{10}$$

where J is the total number of constraints, K_j^c is the coefficient used to scale the constraint violation of the j^{th} constraint, and ε_j is the violation of the j^{th} constraint. The total change in mass target (*ΔM*) is bounded to allow gradual changes in the structure.

6.2.9. State Update Rules

This is the heart of topology optimization method. In this step, the state of a variable is updated based on the state of its neighbors. The state update is carried out in two steps:

1. Field variable update: The field variable (internal energy density) of a variable is updated as accounting for the field variable values of its n neighbors as,

$$U_i = \sum_{j=0}^{n} U_j \Big/ \sum_{j=0}^{n} 1. \tag{10}$$

2. Variable/Material Update: Once the field-variable state of each variable is defined, the design variable is updated to reflect the changes. While numerous rules are proposed in literature [6] to update design variables, a control based rule used by Patel [8] is implemented here (Figure 6-2).

The change in the design variable of i^{th} variable (Δx_i) is computed as,

$$\Delta x_i^t = K\left(U_i^t - U^*\right)/U^*. \tag{11}$$

where K is a scaling factor and U^* denotes the internal energy density set point. The design variable is updated as,

$$x_i^{t+1} = x_i^t + \Delta x_i^t. \tag{12}$$

The change in the variable is constrained by the bounds on the value of the design variable i.e.,

I. if $x_i^{t+1} < LB$, then $x_i^{t+1} = LB$,

II. if $x_i^{t+1} > UB$, then $x_i^{t+1} = UB$,

and only certain discrete values are allowed.

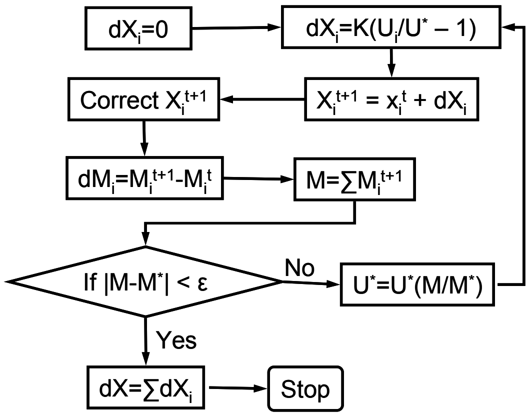


Figure 6-2: Design variable update.

The mass of each element is then calculated by using the appropriate material model associated with the design variables. If the total mass of the structure meets the constraint, the total change in design variables in this iteration is calculated, and the design variable update is considered completed. If the mass constraint is not satisfied, the IED set point is updated iteratively to accommodate the material constraint as,

$$U^* \equiv U^* = U^* M / M^*. \tag{13}$$

where M is the mass of the structure.

6.2.10.Stopping Criteria

Two termination conditions are used to stop the optimization process.

1. The number of iterations has exceeded the maximum number of iterations, or
2. The change in the topology is smaller than the tolerance, i.e.,

$$dX^t = \sum_{i=1}^{N} \Delta x_i^t \le \varepsilon. \tag{14}$$

The numerical oscillations in convergence are limited by averaging the total change in topology over two iterations.

6.3. References

1. A Tovar, *Bone Remodeling as a Hybrid Cellular Automaton Optimization Process*, PhD thesis, University of Notre Dame, 2004.
2. NM Patel, B-S Kang, JE Renaud, Crashworthiness Design using a Hybrid Cellular Automata Algorithm, *In Proceedings of the 2006 International Design Engineering Technical Conference*, DETC-2006-99566, Philadelphia PA, Sep 10-13, 2006.
3. P Hajela, B Kim, On the Use of Energy Minimization of CA Based Analysis in Elasticity, *Structural and Multidisciplinary Optimization*, 23, 23-33, 2001.
4. J Forsberg, L Nilsson, Topology Optimization in Crashworthiness Design, *Structural and Multidisciplinary Optimization*, 33, 1-12, 2007.
5. http://mathworld.wolfram.com, Last accessed 23-March-2008.

6. MP Bendsøe, O Sigmund, Material Interpolation Schemes in Topology Optimization, *Archives of Applied Mechanics*, 69, 635-654, 1999.
7. MP BendsØe, O Sigmund. *Topology Optimization: Theory, Methods and Applications*, Springer-Verlag, Berlin, 1989.
8. NM Patel, *Crashworthiness Design Using Topology Optimization*, PhD thesis, University of Notre Dame, 2004.
9. RT Haftka, Z Gurdal, MP Kamat, *Elements of Structural Optimization*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2nd ed., 1990.
10. SN Patnaik, DA Hopkins, Optimality of Fully-Stressed Design, *Computer Methods in Applied Mechanics and Engineering*, 165, 215-221, 1998.
11. JO Hallquist, *LS-DYNA Manual version 971*, Livermore Software Technology Corporation, October 2007.

7. APPENDIX C: USING A QUEUING SYSTEM

7.1. *Relationship with the LS-OPT queuing system*

This queuing system is the same as used in LS-OPT. If your queue setup works for LS-OPT then it should work for LS-TaSC as well. This appendix mostly repeats the information for people not using LS-OPT.

In the LS-TaSC GUI the queuing is defined in the Scheduling tab of the Case definition. LS-OPT, on the other hand, define the queuing system in the run panel.

Also, you do not need to reinstall the *wrapper* program if it is already installed for LS-OPT.

7.2. *Experience may be required*

Experience with the queuing system and help from the system administer may be required. The queuing systems are not provided by LSTC. Getting the queue system to work may therefore require work and insight from the customer.

7.3. *Introduction*

The LS-TaSC Queuing Interface interfaces with load sharing facilities (e.g. LSF[1] or LoadLeveler[2]) to enable running simulation jobs across a network. LS-TaSC will automatically copy the simulation input files to each remote node, extract the results on the remote directory and transfer the extracted results to the local directory. The interface allows the progress of each simulation run to be monitored via the GUI. The `README.queue` file should be consulted for the most up to date information about the queuing interface.

7.4. *Installation*

To run LS-TaSC with a queuing (load-sharing) facility the following binary files are provided in the `LSOPT_EXE` directory which un-tars (or unzips) from the distribution during installation of LS-OPT:

```
LSOPTOPO_EXE/wrapper
LSOPTOPO_EXE/runqueuer
```

The runqueuer executes the command line for the purpose of queuing and must remain in the LS-TaSC environment (the same directory as the lsopt executable).

The following instructions should then be followed:

[1] Registered Trademark of Platform Computing Inc.
[2] Registered Trademark of International Business Machines Corporation

a) *Installation for all remote machines running LS-DYNA*

1. Create a directory on the remote machine for keeping all the executables including `lsdyna`. Copy the appropriate executable `wrapper` program to the new directory. e.g. if you are running lsdyna on a Linux machine, place the `wrapper` appropriate for the architecture and operating system on this machine. You do not need to reinstall the *wrapper* program if it is already installed for LS-OPT.

b) *Installation on the local machine*

2. Select the queuer option in the GUI for the Case definition.

To pass all the jobs to the queuing system at once, select zero concurrent jobs in the GUI or command file.

In this example, the arguments to the rundyna.hp script are optional and can be hard-coded in the script.

3. Change the script you use to run the solver via the queuing facility by prepending "wrapper" to the solver execution command. Use full path names for both the wrapper and executable or make sure the path on the remote machine includes the directory where the executables are kept.

The argument for the input deck specified in the script must always be the LS-OPT reserved name for the chosen solver, e.g. for LS-DYNA use `DynaOpt.inp`.

7.5. Example

Example: The LS-TaSC command relating to the queue is "/nec00a/mike/project/submit_pbs". The "submit_pbs" file is:

```
#!/bin/csh -f
#
# Run jobs on a remote processor, remote disk
set newdir=`pwd | sed -n 's/.*\/\(.*\)\/\(.*\)/\1\/\2/p'`
# Run jobs on a remote processor, local disk (no transmission)
# set newdir=`pwd`
echo $newdir
cat > dynscr << EOF
#!/bin/csh -f
#
#PBS -l nodes=1:ncpus=1
#
setenv LSOPT /nec00a/mike/codes/LSOPT_EXE
setenv LSOPT_HOST $LSOPT_HOST
setenv LSOPT_PORT $LSOPT_PORT
# Run jobs on a remote processor, remote disk
mkdir -p lsopt/$newdir
cd lsopt/$newdir
# The input file name is required for LS-OPT
```

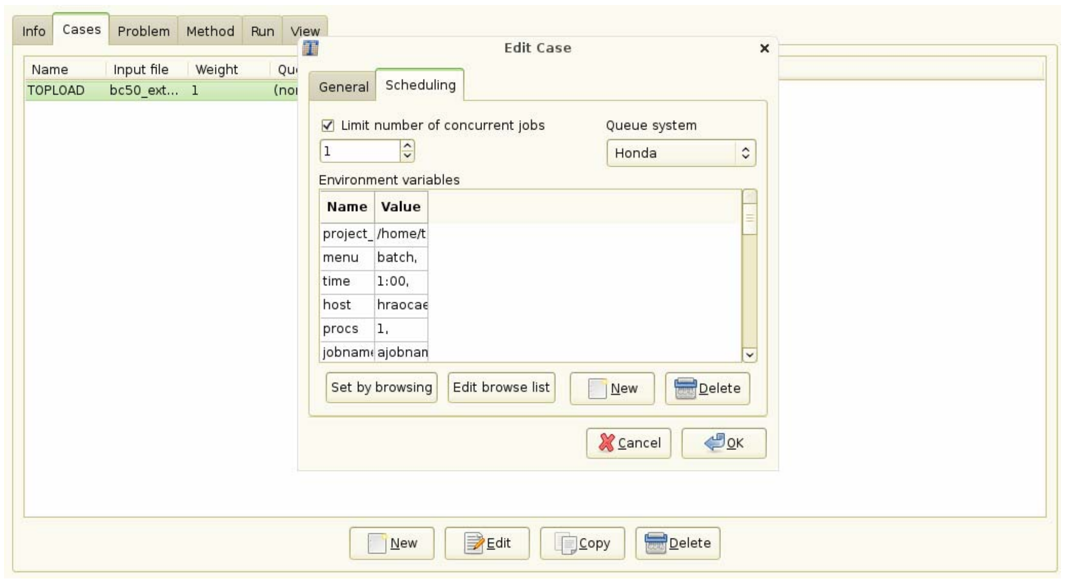
```
/nec00a/mike/codes/wrapper /nec00a/mike/codes/ls980.single
i=DynaOpt.inp
EOF
qsub dynscr
```

It is also possible to specify the queuer command directly on the command line. Environment variables can be specified on the solver command line (e.g. for the PBS queuing system) as well as LS-TaSC input data.

Example:
This example shows how the required environment variables LSOPT_PORT and LSOPT_HOST set by the runqueuer program are specified on the solver command line whereas the two user variables LSDYNA971_MPP and LSOPT_WRAPPER are defined and stored as special input entities (see Section 7.14). These can also be set on the command line using the Linux "`setenv`" command. `qsub` is a PBS queue submit command and the –v directive defined the names of environment variables to be exported to the job. The `qsub` manual pages should also be consulted for more details. Say we submit to qsub using the command "qsub -v LSOPT_PORT,LSOPT_HOST ../../dynscr2". The `dynscr2` file in this case is:

```
# This is the dynscr2 file
#==========================
#!/bin/csh -f
#
#$ -cwd -pe mpi 2
#
setenv NP 2
setenv ROUNDROBIN 0
#
# Define LSDYNA971_MPP environment variables in lsopt input
# or shell command ("setenv").
# $1 represents i=DynaOpt.inp and is automatically
# tagged on as the last argument of the lsopt "solver command".
#
setenv EXE "$LSDYNA971_MPP $1"
#
rm -f mpd.hostfile mpp.appfile
filter_hostfile < $PE_HOSTFILE > mpd.hostfile
#
# This python script builds an HPMPI specific "appfile" telling it
# exactly what to run on each node.
#
gen_appfile.hpmpi mpd.hostfile $SGE_O_WORKDIR $NP $ROUNDROBIN $EXE >
mpp.appfile
#
# This actually executes the job
#
$LSOPT_WRAPPER /opt/hpmpi/bin/mpirun -f mpp.appfile
#
```

The solver command data and environment variable input are displayed below:



Name	Input file	Weight	Qu
TOPLOAD	bc50_ext...	1	(no

[✓] Limit number of concurrent jobs: 1

Queue system: Honda

Environment variables

Name	Value
project_	/home/t
menu	batch,
time	1:00,
host	hraocae
procs	1,
jobname	ajobnan

7.6. *Mechanics of the queuing process*

Understanding the mechanics of the queuing process should help to debug the installation:

1. LS-TaSC automatically prepends `runqueuer` to the solver command and executes runqueuer which runs the `submit_pbs` script.
   - o The `runqueuer` sets the variables `LSOPT_HOST` and `LSOPT_PORT` locally.
   - o In the first example, the `submit_pbs` script spawns the `dynscr` script.
2. In Example 1 the queuing system then submits `dynscr` (see `qsub` command at the end of the submit_pbs script above) on the remote node which now has fixed values substituted for `LSOPT_HOST` and `LSOPT_PORT`.
3. In Example 2 the LS-TaSC schedules the `qsub` command directly with `LSOPT_HOST` and `LSOPT_PORT` as arguments and `i=DynaOpt.inp` appended at the end of the command. It therefore serves as an argument to `dynscr2`.
4. The wrapper executes on the same machine as LS-DYNA, opens a socket and connects back to the local host using the host/port information. The standard output is then relayed to the local machine. This output is also written to the `logxxxx` file (where `xxxx` is the process number) on the local host. To look at the log of any particular run, the user can select a button on the Run page under the *View Log* heading. The progress dialog is shown below, followed by the popup log.

An example of an error message resulting from a mistype of "wrapper" in the submit script is given in another example log file as follows:

```
STARTING command /home/jim/bin/runqueuer
PORT=56984
JOB=LoadLeveler
llsubmit: The job "1/1.1" has been submitted.
/home/jim/LSOPT_EXE/Xrapper: Command not found.
Finished with directory
```

```
/home/jim/LSOPT/4.1/optQA/QUEUE/EX4a_remote/remote/1/1.1
```

5. The wrapper will also extract the data immediately upon completion on the remote node. A log of the database extraction is provided in the `logxxxx` file.

7.7. *Environment variables*

These variables are set on the local side by the `runqueuer` program and their values must be carried to the remote side by the queuing software. The examples above illustrate two methods by which this can be accomplished.
`LSOPT_HOST` : the machine where LS-OPT (and therefore the `runqueuer`) is running.
`LSOPT_PORT` : TCP/IP port `runqueuer` listens on for remote connections

7.8. *Troubleshooting*

1. Diagnostics for a failed run usually appear in the logxxxx file in the run directory. If there is almost no information in this file, the wrapper path may be wrong or the submission script may have the wrong path or permission. For any job, this file can be viewed from the progress dialog on the **Run** page.

   Please attach the log file (lsopt_output) when emailing support@lstc.com.

2. Make sure that the permissions are set for the executables and submission script.

3. Check all paths to executables e.g. "wrapper", etc. No diagnostic can detect this problem.

4. Make sure that the result database is produced in the same directory as where the wrapper is started, otherwise the data cannot be extracted. (E.g. the front end program such as mpirun may have a specification to change the working directory (`-wd dir`)).

5. *Running on a remote disk.* Make sure that the file "`HostDirectory`" is not copied by a user script to the remote disk if the simulation run is done on a remote disk. The "`HostDirectory`" file is a marker file which is present only on the local disk. Its purpose is to inform the wrapper that it is running on the local disk and, if found on a remote disk, will prevent the wrapper from automatically transferring extracted results back to the local disk. In general the user is not required to do any file copying since input files (including LS-DYNA include files) are copied to the remote disk automatically. The response.* and history.* files are recovered from the remote disk automatically.

6. *Termination of user-defined programs:* LS-DYNA always displays a '`N o r m a l`' at the end of its output. When running a user-defined program which does

not have this command displayed for a normal termination, the program has to be executed from a script followed by a command to write `'N o r m a l'` to standard output. The example file *runscript* shown below first runs the user-defined solver and then signals a normal termination.

```
mpiexec -n 2 /home/john/bin/myprogram -i UserOpt.inp
# print normal termination signal to screen
echo 'N o r m a l'
```

which is submitted by the wrapper command in submit_pbs as:

```
/home/john/bin/wrapper /home/john/bin/runscript
```

Note: Adding "`echo N o r m a l`" at the end of the wrapper command (after a semicolon) does not work which is why it should be part of the script run by the wrapper.

7.9. User-defined queuing systems

To ensure that the LS-OPT job scheduler can terminate queued jobs, two requirements must be satisfied:

1. The queuer must echo a string

   `Job "`*`Stringa Stringb Stringc …`*`" has been submitted`
   or
   `Job `*`Stringa`*` has been submitted`
   e.g.
   ```
   Job "Opteron Aqs4832" has been submitted
   Job aqs4832 has been submitted
   ```

   The string will be parsed as separate arguments in the former example or as a single argument in the latter example. The string length is limited to 1024 characters. The syntax of the phrases "`Job `" and " `has been submitted`" must be exactly as specified. If more than one argument is specified without the double quotes, the string will not be recognized and the termination feature will fail.

2. A termination script (or program) `LsoptJobDel` must be placed either in the main working directory (first default location) or in the directory containing the LS-OPT binaries (second default). This script will be run with the arguments *stringA, stringB,* etc. and must contain the command for terminating the queue. An example of a Unix C shell termination script that uses two arguments is:

```
#!/bin/csh -f
aadmin -c $1 -j $2 stop
```

7.10. Blackbox queueing system

The Blackbox queueing system is another flavor of the User-defined queueing system. It can be used when the computers running the jobs are separated from the computer running LS-OPT by means of a firewall. The key differences between User-defined and Blackbox are:

1. It is the responsibility of the queueing system or the user provided scripts to transfer input and output files for the solver between the queueing system and the workstation running LS-OPT. LS-OPT will not attempt to open any communications channel between the compute node and the LS-OPT workstation.
2. Extraction of responses and histories takes place on the local workstation instead of on the computer running the job.
3. LS-OPT will not run local placeholder processes (i.e. extractor/runqueuer) for every submitted job. This makes Blackbox use less system resources, especially when many jobs are run in each iteration.

When using the Blackbox queueing system, a `LsoptJobDel` script is required, just as in the User-defined case. Furthermore, another script named `LsoptJobCheck` must also be provided. This script takes one parameter, the job ID, as returned by the submission script. The script should return the status of the given job as a string to standard output.

The Blackbox queuer option requires the user to specify a command that will queue the job. The Blackbox option can also be specified in the Scheduling panel when defining a Case. The command to queue the job must return a *job identifier* that has one of the following two forms:

```
Job "Any Quoted String" has been submitted
Job AnyUnquotedStringWithoutSpaces has been submitted
```

The Word "`Job`" must be the first non-white space on the line, and must appear exactly as shown. Any amount of white space may appear between "`Job`" and the job identifier, as well as after the job identifier and before "`has been submitted`".

The Blackbox queuer requires the presence of two executable scripts `LsoptJobCheck` and `LsoptJobDel`. These scripts must be located in either in the current LS-OPT project directory or in the directory where the running LS-OPT program is located. (For Windows, the scripts must have an added extension `.exe`, `.vbs`, `.cmd` or `.bat`). If the Blackbox queuer option is invoked for some solver, then LS-OPT checks for the existence of executable scripts in one of these locations, and refuses to run if the `LsoptJobCheck` and/or `LsoptJobDel` scripts cannot be found or are not executable. The project directory is searched first.

LsoptJobCheck script

The user-supplied `LsoptJobCheck` script is run each time LS-OPT tries to update the current status of a job. The `LsoptJobCheck` script is run with a single commandline argument:

```
LsoptJobCheck job_identifier
```

The working directory of the `LsoptJobCheck` script is set to the job directory associated with job_identifier.

The script is expected to print a status statement that LS-OPT can use to update its status information. The only valid status statements are:

String	Description
`WAITING`	The job has been submitted and is waiting to start
`RUNNING`	The job is running.
`RUNNING` *`N/M`*	After `RUNNING`, the script may also report the progress as a fraction. `RUNNING 75/100` means that the job has ¼ to go. The progress information will be relayed to the user, but not used in any other way by LS-OPT.
`FAILED`	The job failed. This is only to be used when the underlying queueing system reports some kind of problem. Hence, a solver that has terminated in error does not have to be deteceted by the `LsoptJobCheck` script.
`FINISHED`	The job has completed and any output files needed for extraction has been copied back to the run directory.

Any amount of white space may appear at the beginning of a status statement, and anything may appear after these statements. The optional *N/M* argument for `RUNNING` is interpreted as an estimate of the progress; in this case *N* and *M* are integers and *N/M* is the fractional progress. *N* must be not be larger than *M*.

If `LsoptJobCheck` terminates without printing a valid status statement, then it is assumed that `LsoptJobCheck` does not function properly, and LS-OPT terminates the job using the `LsoptJobDel` script. All output from the `LsoptJobCheck` script is logged to the job log file (`logxxxx`) in the run directory for debugging purposes.

Note: The `LsoptJobCheck` script may print more than one status statement, but only the first one will be used to update the status.

LsoptJobDel script

The user-supplied `LsoptJobDel` script is run whenever the user chooses to terminate a job, or whenever LS-OPT determines that a job should be killed (for example, if `LsoptJobCheck` fails). The `LsoptJobDel` script is run with a single commandline argument:

```
LsoptJobDel job_identifier .
```

The working directory of the `LsoptJobDel` script is set to the job directory associated with job_identifier.

7.11. *Honda queuing system*

The Honda queuing system interface is based on the Blackbox queuing system, but is dedicated to the particular needs of this system.

Mechanics of the Honda queuing process

The queuing system generates a status file for which an environment variable has been defined in LS-OPT as:

```
$HONDA_STATUSFILE
```

The status file is the output of the PBS queue check command. During the initialization phase, LS-OPT checks whether this variable setting points to a valid file. If it does not, LS-OPT terminates before starting the scheduler, and prints a standard LSOPT-style error message.

The line which marks the fields in the status file is used to determine how to parse the file; this line has the form "----- ---------- - ----- ---- ....". Fields are extracted based on this line which consists solely of space and dash characters. The following fields are used:

4	name
6	status: 'R' for running or 'Q' for queued
10	total wall clock time allowed
11	total wall clock time consumed.

Fields 10 and 11 are used to set the progress indicator. If the indicator ever reaches 100%, then it will terminate due to total wall clock time restrictions.

If a job cannot be found in the status file, then it is assumed to be dead. The job status entry is not looked for until a minimum of 3 seconds after the job has been started. A status file is searched for a particular job status entry only if the status file has a modification time that is later than the start time of the job.

Since there is no way to determine the exit status of a job by looking only at this status file, the determination of the final exit status depends on whether or not the job is an LS-DYNA job. If the job is an LS-DYNA job, then the messag file is parsed for the status statements "N o r m a l" and "E r r o r" termination. If no messag file is found *10 seconds* after the job is no longer listed in the status file, then we assume an error termination.

If the job is a non-LS-DYNA job, then LsoptJobCheck (see Section 7.10) is executed just once after the job no longer appears in the status file. LsoptJobCheck should print either (a) FINISHED or (b) ERROR in order to communicate the final exit status. If LsoptJobCheck cannot be found or cannot be executed, then ERROR is assumed. The job log file will contain a message indicating any problem that may exist which prevents LsoptJobCheck from being run.

The HONDA queued jobs do not use LsoptJobDel as defined in the Blackbox queuing selection. Jobs are deleted using the standard PBSPro qdel command.

Various statements concerning how status information is gathered are logged to the job log files. These are:

1. Job status for LSDYNA jobs found in 'messag' file:

```
[HONDA] Termination status found in 'messag' file
[HONDA] exact termination statement
```

2. The job status line for the current job found in $HONDA_STATUSFILE is saved:

```
[HONDA] status line
```

3. The job is assumed finished if there is no status line found:

```
   [HONDA] Job 23551 not found in STATUS file - assuming job is
finished.
```

4. Indication that LsoptJobCheck is run at the end of a non-LS-DYNA job:

```
   [HONDA] Non LS-DYNA job. Running LsoptJobCheck to determine
exit status.
```

5. Status returned from LsoptJobCheck.

```
   [HONDA] Job finished - LsoptJobCheck reports normal
termination
  [HONDA] Job finished - LsoptJobCheck reports error termination
```

Any errors while gathering status information are logged to the job log files such as log12345.

6. Missing messag file after LSDYNA terminates:

```
[HONDA] Failed to find 'messag' file while FINISHING.
[HONDA] Assuming ERROR termination for LSDYNA job.
```

7. Found no termination status statement in messag file

```
[HONDA] Found no termination status in 'messag' file
[HONDA] Assuming ERROR termination for LSDYNA job.
```

8. HONDA_STATUSFILE variable not set

```
[HONDA] *** Error $HONDA_STATUSFILE not set.
```

9. Could not open $HONDA_STATUSFILE

```
[HONDA] *** Error Failed to open $HONDA_STATUSFILE=pbsq_status
```

10. LsoptJobCheck script not found for non-LSDYNA job

```
[HONDA] *** Error LsoptJobCheck cannot be found.
[HONDA]     Assuming error termination for non-LSDYNA job.
```

11. LsoptJobCheck script did not print either (a) FINISHED or (b) FAILED.

```
   [HONDA] *** Error LsoptJobCheck did not return a valid status.
   [HONDA]          Assuming error termination for non-LSDYNA
job.
```

If $HONDA_STATUSFILE is not updated in a timely fashion, then the scheduler can hang forever, never moving forward. A message is passed to lsopt through the communication socket if this happens:

```
 *** Warning HONDA_STATUSFILE out of date by more than 5 minutes
 *** Job progress monitoring suspended until next update
```

Even though the status file is checked before starting the scheduler, it is still possible for file errors to occur. These are also sent directly to LS-OPT.

```
 *** Error $HONDA_STATUSFILE not set
 *** Error Failed to open $HONDA_STATUSFILE=pbsq_status
```

7.12. Microsoft Windows Compute Cluster server

LS-OPT supports submission of jobs to the Microsoft Compute Cluster Pack Scheduler. Two scripts called `submit.cmd` and `submit.vbs,` that work together, are available to interface LS-OPT with CCP. The script can be downloaded from `ftp://ftp.lstc.com/ls-opt/QUEUING/MSCCS`. Before using the scripts the variables in the beginning of the file `submit.cmd` needs to be changed to fit your local environment. Most users do not need to change the `submit.vbs` file.
The example shows how the queue-related parts of an LS-OPT command file look when using the CCP scripts, when they are placed in the same directory as the command file:

7.13. Simple manual setup for running LS-OPT and solvers on different machines

A convenient setup is one in which LS-OPT runs on e.g. a Windows machine and the solvers are running on a cluster (typically Linux). Such a setup can be created as follows:

1. Install LS-OPT on a Windows (or any desired) machine for preparing the input. Create the problem setup using LS-OPTui. The `solver command` should be created for running jobs on a cluster. This can be done by selecting any of the queuing systems supported by LS-OPT or, if all the jobs will be running on the same cluster where LS-OPT resides, simply by specifying the solver executable name as a *solver command.* The number of concurrent jobs should be set for each case on the **Run** page. Save the input to a file using e.g. the name `com.`
2. Open the `com` file with LS-OPTui and create a second command file e.g. `com.pack` by selecting **Tools→Gather LS-OPT database + histories/responses** and saving as `com.pack`. There are now two command files: `com` for running the optimization task and `com.pack` for packing the output data after the run.
3. Install LS-OPT and the solver executables on a cluster node for running LS-OPT in batch mode. Copy the recently created problem setup with the two command files from the Windows machine onto a cluster node. This setup should allow the user to run an LS-OPT job in batch mode.
4. Run LS-OPT by executing the command: `lsopt com` on the cluster. This is done from the command line.

5. After completion of the LS-OPT run, execute `lsopt com.pack` to create a file `lso_pack.tar.gz` containing the entire run database.
6. Unzip `lso_pack.tar.gz` on the Windows machine to do the post-processing.

7.14. *Passing environment variables*

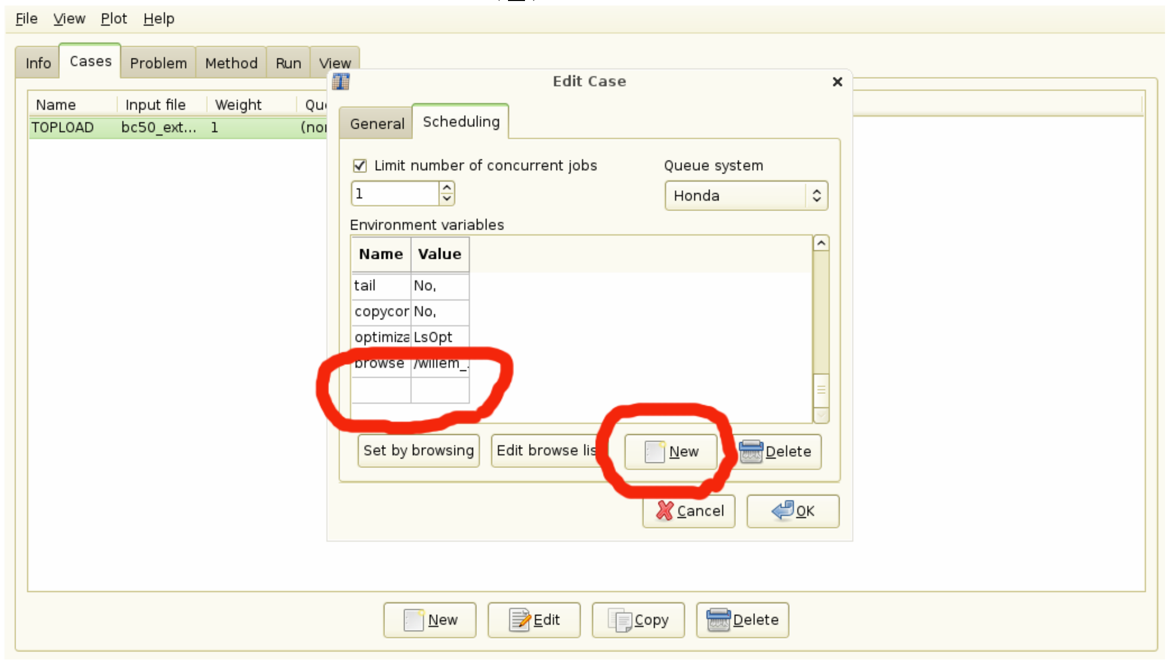
LSOPT provides a way to define environment variables that will be set before executing a solver command. The desired environment variable settings can be specified directly in the com file with solver commands:

They can be specified within the Scheduling tab when defining a Case.

7.14.1.Adding a new environment variable definition

Select the New button. After selecting this option, an empty, editable environment variable definition will appear.

We do not allow the names of variables to contain anything other than upper- or lower-case letters, numbers, and underscore (_) characters. Variable values are not so limited.



Name	Input file	Weight	Qu
TOPLOAD	bc50_ext...	1	(no

☑ Limit number of concurrent jobs: 1

Queue system: Honda

Environment variables

Name	Value
tail	No,
copycor	No,
optimiza	LsOpt
browse	/willem_

7.14.2.Editing an existing environment variable definition

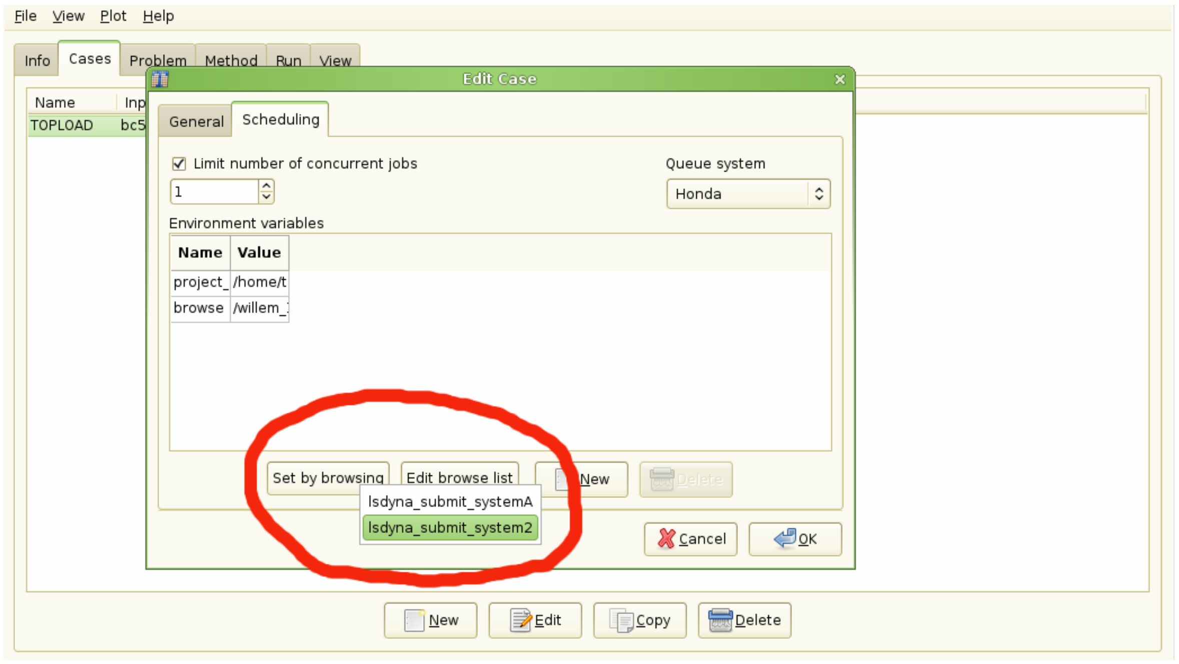
To edit an environment variable, double-click on the environment variable in the Environment variables list. The display mode of the variables will change to make it editable.

7.14.3.Set by browsing

Select the **Select by Browsing** button. In order for this option to work, user-supplied executables must be present in the directory
$HOME/LSOPT_SCRIPTS
The directory LSOPT_SCRIPTS must exist as a subdirectory of the user's home directory, and it must contain executables. If the directory LSOPT_SCRIPTS does not exist, or if there are no executables in this directory, an error box will appear. Setting the LSOPT_SCRIPT Unix/Linux/Windows system environment variable may specify an alternative script directory.
After selecting the **Set by browsing** option, a dialog of buttons will appear, one for each executable in this directory. For example, suppose this is the directory listing for `$HOME/LSOPT_SCRIPTS`:

```
-rwxr-xr-x 1 joe staff 13597 2009-12-01 18:09 lsdyna_submit.autounion*
-rw-r--r-- 1 joe staff 13597 2009-12-01 17:46 stdin.save
-rwxr-xr-x 1 joe staff     9 2009-08-10 14:23 test*
-rwxr-xr-x 1 nielen staff  9 2009-08-10 14:26 testb*
```

Then, when you select the Set by browsing option, the following dialog appears:



A valid browse command must print environment variable definitions to standard output in the form name='value'; the single quotes are optional if value does not contain spaces. A valid sample output is shown below (the line is wrapped because of its length).

```
exe=/home/trent/LSTC/PERL/lsdyna-caec01_pbs_sub.pl menu=batch
time=1:00 host=abcdefgh07 procs=1 jobname='My Job' project=isd
email=No delay=No preemptable=No version='LS-DYNA 970 MPP SP
6763' ioloc=/home/trent inpfile=DynaOpt.inp mem1=auto mem2=auto
```

```
pfile=Generic dumpbdb=No dynamore=No clean=No tail=No copycont=No
optimization=LsOpt
```

All of the name='value' strings are directly imported into the Env Vars tab in bulk. In addition to these **Browse List** variables, a special browse variable is created that should not be edited. This variable records the program name used to create the Browse List.

NOTE: All variables must be printed on one line, which must be the last line of output from the program. Lines before the last line are ignored.

WARNING: The user-supplied browse program should never define the browse variable in its output. The name `browse` should be treated as a reserved name.

A simple Linux browse command could be a shell script:

```
#!/bin/bash
echo This line is ignored. Only the last line survives
echo A=B C=D
```

Running the browse command shown above will import two variables, A and C, into the browse list.

NOTE: Strings in the Env Vars List appearing above the browse= line are all part of the Browse List. Strings in the Env Vars tab that appear below browse= are never part of the Browse List. User-defined environment variables will always follow after the browse variable definition (e.g., last=first in the figure above was not defined by the browse command.)

7.14.4.Edit browse list

Select the Edit Browse list button. Choosing this option does nothing unless a Browse List has been previously created. If a valid Browse List is present in the Env Vars tab, then selecting this option will run the original program that created the Browse List, together with all of the current Browse List options passed as command line arguments, one per existing environment variable. Each command-line argument has the form name=value. However 'value' is not single-quoted because each name=value argument is a separate command-line argument. The customer-supplied browse command should offer the user an opportunity to edit the existing variables, and the browse command should return the newly edited list on one line, in the same format as described above. This would normally be done through some sort of graphical user interface. The returned list will be used to replace all of the previous Browse List.
The next example script returns an initial Browse List consisting of two variables, A and C. Invoking the editing feature appends a new variable (tN=N) to the list.

```
#!/bin/bash
echo This line will be ignored. Only the last line survives.
if [ "$1" == "" ]; then
  echo A=B C=D;
else
  echo $* "t"$$"="$$;
fi
```

When this script is invoked using the “Create by Browse” feature, there are no command-line arguments, and the script prints “A=B C=D” to standard output. However, when the script is invoked using the edit feature for the first time, two command-line arguments “A=B” and “C=D” are passed to the script. This time the return line consists of the original command-line arguments (printed using $*) and tN=N, where N is the PID of the shell process. If the editing feature is invoked a second time, then three command-line arguments are passed to the script (“A=B”, “C=D”, and “tN=N”). Another new variable tN is appended, where N is the newest PID of the script process. This sample script has little practical value, except to illustrate how existing variable settings are passed by command-line to the previous browse command, and to illustrate how one can use the editing feature to modify or add new variables.
Note: The browse command can ABORT the replacement operation by printing a blank line to the standard output and immediately terminating. Otherwise the current Browse List may be deleted. If the browse command abnormally terminates, then an error box will appear with a title bar indicating that the command failed.

7.14.5.How the browse list is used by LS-TaSC

The **Browse List** (indeed, the complete **Env Vars List**) is used to set environment variables before running the solver command specified by LSOPT. However, if the first variable returned by the browse command is **exe**, then a pre-processing command is run before running the actual solver command. The pre-processing command is the value of the **exe** variable. The pre-processing command has a command line

```
$exe var1=$var1, var2=$var2, ... varN=$varN
```

That is, the command executed is the value of the **exe** variable; additional command line arguments consist of all **Browse List** strings with a comma delimiter appended to each intermediate one. (The final argument is not followed by a comma.)
Note: Such a pre-processing command is always run from within the current **LSOPT Job Directory**. Therefore, any file that the pre-processing command references must be specified by a fully-qualified path or must be interpreted relative to the current **LSOPT Job Directory**. So, the **LSOPT Case Directory** will be **".."** and the **LSOPT Project Directory** will be **"../..”.**