The LS-TaSC™ Tool

TOPOLOGY AND SHAPE COMPUTATIONS

SCRIPTING MANUAL

August 2020

Version 4.2

Corporate Address

Livermore Software Technology, an Ansys Company

P. O. Box 712

Livermore, California 94551-0712

Support Addresses

Livermore Software Technology, an Ansys Company

7374 Las Positas Road

Livermore, California 94551

Tel: 925-449-2500 ◆ Fax: 925-449-2507

Livermore Software Technology, an Ansys Company

1740 West Big Beaver Road

Suite 100

Troy, Michigan 48084

Email: sales@lstc.com

Website: www.lstc.com

Tel: 248-649-4728 ♦ Fax: 248-649-6328

Disclaimer

Livermore Software Technology, an Ansys company reserves the right to modify the material contained within this manual without prior notice.

The information and examples included herein are for illustrative purposes only and are not intended to be exhaustive or all-inclusive. Livermore Software Technology, an Ansys company assumes no liability or responsibility whatsoever for any direct of indirect damages or inaccuracies of any type or nature that could be deemed to have resulted from the use of this manual.

3-Aug-20

1. 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:

1. Define the problem and methodology data
2. Call the topology design functions

1.1. 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.

1.2. Script Execution

A script can be invoked from the GUI or the command line.

From the GUI use the *Script->Play script* pull-down to invoke a script. A script submitted from the GUI can use the database that is already open in the GUI. Example 1.6.3 at the end of this chapter demonstrates. Note that the GUI actually issues scripting commands for all actions affecting the database which can be viewed using the *Script->Console* pull-down.

A script can be invoked from the command prompt by running the executable *lstasc_script*. The input command file (script) can be supplied as: With the execution command and a script file name `$ lstasc_script lst_script.lss`.

1.3. Data-structures

1.3.1. 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;

}
```

1.3.2. 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;

  Int   DeleteShells;

  Int   ExitOnNegEID;

  Float OptDeckUserPercent;

  struct lst_DoEData * DoEData;

}
```

`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* ≤ 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.

`DeleteShells`: Set this to a non-zero value (default) to delete shells elements with a thickness less than the minimum specified for the part.

1.3.3. 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;

  Int CaseWeighing;

}
```

`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.

`CaseWeighing`: Set to 1 for using the static weighing (the default), or set to 2 to use dynamic weighing.

1.3.4. 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.

1.3.5. lst_Surface

The details of a surface are:

```
struct lst_Surface {Int ID;
                        Int Objective;
                        Float ProxTol;
                   Float VarMoveLimit;
                 Float TargetField;
                 struct lst_Geometry * GeometryList;
                   Float RemeshRelativeDepth;
                 struct lst_Part * Next; }
```

`ID`: Each surface is identified with a unique id for the *SET_SEGMENT as in the LS-DYNA input deck.

`Objective`: The objective for designing the surface. This will be used to select the target value of the field of the surface. 1 = match the average, 2 = minimize stress, 3 = minimize volume, and 4 = match the target value. The default is to match the average value over the surface.

`ProxTol`: All elements within a radius of proximity tolerance would be considered as the neighbors of an element. The default of -1 will prompt the program to compute a suitable value.

`VarMoveLimit`: Maximum change of a nodal location. The default of -1 will prompt the program to compute a suitable value.

`TargetField`: Target value to be used if Objective = 4.

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

`RemeshRelativeDepth`: The depth of the remeshing in terms of the times the average segment side length. The default is 4.

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

1.3.6. lst_Geometry

The details of a geometry definition are:

```
struct lst_Geometry {  Char *Name;
                       Int Type;
                       Int CID;
                       Int Set;
                       Int ExtructionDir;
                       Int MirrorPlane;
                       Float ForgeThick;
                       Float SplineWidth;
                      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, 5 is a double sided casting constraint, and 6 is a forging.

`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.

`ForgeThick`: The thickness of a forging definition.

`SplineWidth`: The width of a spline edge interpolation in surface design.

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

1.3.7. 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_DynWeight   *DynWeight;
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.

1.3.8. lst_Constraint

The structural constraints for a load case 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.

1.3.9. lst_DynWeight

The dynamic weighing of a load case is specified in the following data structure:

```
struct lst_DynWeight {
    Char * ConstraintName;
    Float Scale;
    Float Offset;
}
```

`ConstraintName`: The name of the constraint.

`Scale`: The scaling of the constraint value.

`Offset`: The offset to be added to the constraint value.

1.3.10. lst_DoEData

This data structure contains the multi-point information.

```
struct lst_DoEDatat {
    Int UseDoE;
    Int DesignStrategy;
    Int Freq;
    Float TimeConstant;
    Int MultiPointStrategy;
    Int NumPoint;
    Char * PartSubregionSize;
    Char * WeigthRatioSubRegionSize;
}
```

`UseDoE`: This parameter indicates whether to use the multi-point scheme. A value of zero indicates not to use it.

`DesignStrategy`: This parameter indicates which design strategy to use.

`Freq`: This parameter indicates whether how often to compute the derivatives.

`TimeConstant`: This parameter is a time constant for the filter of the derivatives over multiple iterations.

`MultiPointStrategy`: This parameter indicate how to select the points.

`NumPoint`: The number of points in the multi-point strategy.

`PartSubregionSize`: This parameter indicates the computation of the move limits on the part mass fractions.

`WeightRatioSubregionSize`: This parameter indicates the computation of the move limits on the load case weight ratios.

1.3.11. 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.

1.4. 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.

1.4.1. Definition

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

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

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

1.4.2. Initialization

During initialization, the user provides the necessary input data.

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);

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);
```

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 );
```

<u>Example: Adding two constraints to a case</u>

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.

Adding dynamic weighing of the load cases

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

```
struct * lst_DynWeight lst_CaseAddDynWeight ( struct lst_Case* cse, Char * constraintName, Float Scale, Float Offset );
```

Example: Adding dynamic weighing to a case

```
root->Problem->CaseWeighing = 2;

lst_CaseAddDynWeight ( aCase, "gDisp", 1.0, 0.0 );
```

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 );
```

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 );
```

Adding Surface Data

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

```
struct lst_Surface * lst_ProblemAddSurface ( struct lst_Problem *prob, Int segmentId, Int objective, Float
```

```
varMoveMax, Double proxTol, Double targetField, Double
relativeMeshDepth );
```

with the items in the command as explained for the Surface structure. The last four arguments (the minimum variable value, the neighbor radius, the target value, and meshing depth) are optional.

Example: Adding a shape

```
struct lst_Surface * srf = lst_ProblemAddSurface ( root-
>Problem, 102, 4, 10., 0.3, 200e6 );
```

Accessing a Surface

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

```
lst_Surface * surface1 = root->Problem->SurfaceList;

lst_Surface * surface2 = root->Problem->SurfaceList-
>Next;

lst_Surface * surface4 = surface1->Next->Next->Next;

lst_Surface * prt = lst_ProblemAddSurface( root-
>Problem, 101, 1 );
```

Adding Geometry Data

Add geometry constraints to the part using the following commands:

```
struct lst_Geometry *

lst_PartAddGeometryExtrusionDir( struct lst_Part *,  Char
* name, Int dir, Int CID );

struct lst_Geometry * lst_PartAddGeometryExtrusionSetDir(
struct lst_Part *,  Char * name, Int set, Int dir, Int
CID );

struct  lst_Geometry  *  lst_PartAddGeometryExtrusionConn(
struct lst_Part *, Char * name, Int set );

struct   lst_Geometry   *   lst_PartAddGeometrySymmetryXY(
struct lst_Part *, Char * name, Int CID );

struct   lst_Geometry   *   lst_PartAddGeometrySymmetryYZ(
struct lst_Part *, Char * name, Int CID );

struct   lst_Geometry   *   lst_PartAddGeometrySymmetryZX(
struct lst_Part *, Char * name, Int CID );

struct  lst_Geometry  *  lst_PartAddGeometry1SideCasting(
struct lst_Part *, Char * name, Int dir, Int CID );

struct  lst_Geometry  *  lst_PartAddGeometry2SideCasting(
struct lst_Part *, Char * name, Int dir, Int CID );
```

Add geometry constraints to the surface using the following commands:

```
struct lst_Geometry *

lst_SurfaceAddGeometryExtrusionDir( struct lst_Surface *,
Char * name, Int dir, Int CID );

struct lst_Geometry * lst_SurfaceAddGeometrySymmetryXY(
struct lst_Surface *, Char * name, Int CID );

struct lst_Geometry * lst_SurfaceAddGeometrySymmetryYZ(
struct lst_Surface *, Char * name, Int CID );

struct lst_Geometry * lst_SurfaceAddGeometrySymmetryZX(
struct lst_Surface *, Char * name, Int CID );

struct lst_Geometry *

lst_SurfaceAddGeometryNewEdgeAuto( struct lst_Surface *,
Char * name, Float width );

struct lst_Geometry *

lst_SurfaceAddGeometryNewEdge( struct lst_Surface *, Char
* name, Int nodeSetID, Float width );
```

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 );
```

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;`

1.4.3. Execution Functions

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" );
```

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" );
```

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);
```

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.

1.5. 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).

1.6. Example Script

1.6.1. 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 );
```

1.6.2. 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);
```

1.6.3. Adding a geometry definition using a script from the GUI

This example adds a specialized extrusion definition to a database opened in the GUI. The script is submitted from the GUI and the script therefore uses the database that is already open. The extrusion definition is added to the first design part in the database.

```
lst_Root *root = _guiroot1;

lst_Part *part = root->Problem->PartList;

// part = part->Next; // if needed follow linked list to correct part

if( part ) {

    printf( "Adding geom def to part %d\n", part->ID );

    lst_PartAddGeometryExtrusionSetDir( part, "Extrusion with dir specified", 1, 1, 0 );

} else {

    printf( "Part not found\n" );

}
```

1.6.4. Creating a topology database

The example performs topology optimization of a single load case problem using extrusion mode.

```
lst_Root *root = lst_RootNew();

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 );
```

1.6.5. 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, "" );
```

1.6.6. Printing the content of the results database

```
Int flag, numV, iter = 1;
Char dirName[1024];
Float *data, *aveChng;
print( "\"ItNum\",\"Mass_Redistribution\"\n" );
Int handle = lsda_open( "lst.binout" );
lsda_cd( handle, "/" );
sprintf( dirName, "/design#%d", iter );
while ( lsda_cd( handle, dirName ) == 1 ) {
  aveChng = lsda_getR8data( handle, "Mass_Redistribution", &numV );
  print( iter, ", ",  aveChng[0], "\n");
  iter = iter+1;
  sprintf( dirName, "/design#%d", iter );
  free( aveChng );
}
lsda_close( handle );
```