SANDIA REPORT

SAND2018-10815
Unlimited Release
Printed 09/25/2018

LDRD Final Report: Topology Optimization for Nonlinear Transient Applications Using a Minimally Invasive Approach

Joshua Robbins

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2018-10815
Unlimited Release
Printed 09/25/2018

LDRD Final Report: Topology Optimization for Nonlinear Transient Applications Using a Minimally Invasive Approach

Joshua Robbins

Abstract

The purpose of this project was to devise, implement, and demonstrate a method that can use Sandias existing analysis codes (e.g., Sierra, Alegra, the CTH hydro code) with minimal modification to generate objective function gradients for optimization-based design in transient, non-linear, coupled-physics applications. The approach uses a Moving Least Squares representation of the geometry to substantially reduce the number of geometric degrees of freedom. A Multiple-Program Multiple-Data computing model is then used to compute objective gradients via finite differencing. Details of the formulation and implementation are provided, and example applications are presented that show effectiveness and scalability of the approach.

Contents

List of Figures

Chapter 1

Introduction

Gradient-based optimization methods, such as topology optimization (TO), are proving effective for engineering design in linear, static applications. For these applications, robust and practical methods are available for computing the needed objective function gradients. However, current methods for computing objective function gradients are impractical for transient, non-linear, coupled-physics applications.

The purpose of this work was to develop and evaluate an alternative approach for generating design sensitivities, i.e., objective function gradients. This approach consists of two key components: 1) a Moving Least Squares (MLS) representation of the design geometry is used to produce far fewer optimization degrees of freedom (dofs): $10^2 - 10^3$ versus $10^4 - 10^6$ in standard topology optimization. This is the core idea of this work. The significantly reduced set of optimization degrees of freedom permits calculation of the objective function gradient by finite differencing, i.e., by varying each dof separately then performing the simulation on the perturbed design to determine the change in the objective function. 2) The Multiple-Program, Multiple-Data (MPMD) functionality available in Sandia's Plato Engine [3] is used to compute the derivative of the objective function with respect to each design variable simultaneously. Each individual perturbation simulation can use multiple processors, however no communication is required between the perturbations, so scaling on current and emerging high performance computing (HPC) platforms is excellent.

The method has two principle advantages over current approaches: 1) It is non-invasive. Because the integrated code is used to evaluate samples at points in the design space, relatively little modification is needed to achieve gradient-based design. This is in contrast to current methods which require significant modification particularly in non-linear, transient physics codes. 2) It uses computing resources very efficiently. The Plato Engine and MPMD approach automate finite differencing of the objective with respect to the chosen design space and allow the perturbation calculations to be run simultaneously. Because the individual calculations are independent, parallel scaling is excellent. While this approach could be characterized as brute force, it may be an efficient and effective use of existing (legacy) simulation codes on emerging, extreme-scale HPC platforms.

Chapter 2

Methods

Moving Least Squares (MLS)

The geometry is defined by a set, $\mathcal{P}$, that comprises values, $p = \{p_i, i = 1...N_p\}$, at control points, $\boldsymbol{\xi}_i \in \mathbb{R}^n$:

$$\mathcal{P} = \{(\boldsymbol{\xi}_i, p_i) \,|p\,(\boldsymbol{\xi}_i) = p_i, i = 1...N_p\} \tag{2.1}$$

where N_p is the number of control points in the MLS representation. The moving least squares function, $f(\mathbf{x})$, is defined implicitly on a domain, Ω, as the locus of minima:

$$\min_f \left(\sum_{i=1}^{N_p} (f - p_i)^2 \exp\left(- \left(\frac{||\mathbf{x} - \boldsymbol{\xi}_i||}{r} \right)^2 \right) \right) \;\; \forall \; \mathbf{x} \in \Omega \tag{2.2}$$

The interaction radius, r, determines the domain of influence of the individual MLS points. A closed form expression for the MLS function is found by differentiating the implicit form in Equation (2.2) with respect to f and equating to zero:

$$f(\mathbf{x}; p) = \frac{\sum_{i=1}^{i=N_p} p_i \exp\left(- \left(\frac{||\mathbf{x} - \boldsymbol{\xi}_i||}{r} \right)^2 \right)}{\sum_{j=1}^{j=N_p} \exp\left(- \left(\frac{||\mathbf{x} - \boldsymbol{\xi}_j||}{r} \right)^2 \right)} \tag{2.3}$$

The geometry can be defined by the level surface at zero:

$$\Omega = \{\mathbf{x} \mid \mathbf{x} \in \mathbb{R}^n, f(\mathbf{x}) \geq 0\} \tag{2.4}$$

$$\partial\Omega = \{\mathbf{x} \mid \mathbf{x} \in \mathbb{R}^n, f(\mathbf{x}) = 0\} \tag{2.5}$$

In this work, the control points are fixed, so the geometry is completely defined by the values, p.

Different methods are available for defining the geometry in a finite element discretization. A straightforward approach is to compute the values of the MLS function at each node in the mesh

$$\rho = \{f(\mathbf{x}_i; p), i = 1...N_n\} \tag{2.6}$$

and use a penalty function to reflect the presence or absence of material at a point. For this work, the geometry is defined on the mesh by using a regularized Heaviside function, $H(\rho)$, to project the MLS into the range $[h_{min}, 1]$ over the regularization length, α [1]:

$$H(\rho) = \begin{cases} h_{min} & \rho < -\alpha \\ h_{min} + (1 - h_{min}) \left(\frac{1}{2} \left(1 + \sin(\frac{\pi\rho}{2\alpha}) \right) \right)^{\beta} & -\alpha < \rho < \alpha \\ 1 & \rho > \alpha \end{cases} \tag{2.7}$$

The regularized Heaviside function is shown in Figure (2) for regularization length, α, of 0.1 and various penalty values, β.

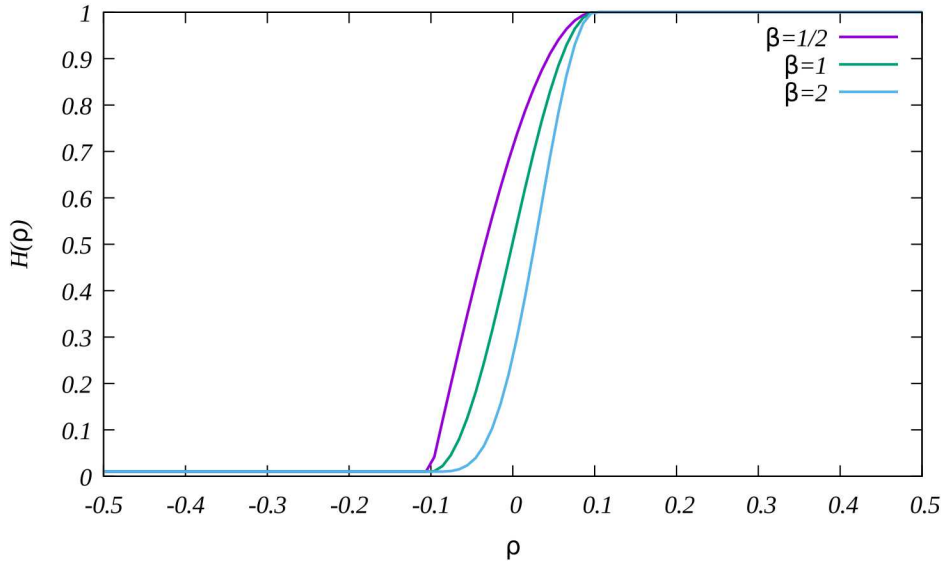


Figure 2.1. Regularized Heaviside functions for various penalty values.

MLS implementation

The central purpose of this work was to evaluate the viability of an MLS geometric representation for topology optimization and to permit continued reseach and development in methods that leverage MLS geometry. To support these future efforts, the MLS functionality described above was implemented in the Plato Engine with a moderate level of software development rigor (i.e., inline documentation, unit testing, and regression testing). This section provides a brief description the implementation. For information on the unit tests and regression tests go to gitlab.com/platoengine.

MLS Operations

The MLS functionality is implemented in a class called `MovingLeastSquares` that uses hardware abstractions available in Kokkos (github.com/kokkos) to ensure portability. The results shown in this report were generated using Graphics Processing Units (GPUs), but the code compiles on a variety of hardware architectures.

Input parameters in xml format are used to construct a `MovingLeastSquares` instance with the desired size, dimensions, etc. Once an instance is defined (see the unit tests for details on input), the operations below are available as member functions.

Compute MLS function values: Evaluate the MLS function given the control point values, p, and evaluation points, $\mathbf{x}_j$:

$$F = \{f\left(\mathbf{x}_j; p\right), j = 1...N_n\} \tag{2.8}$$

Map derivative: Map a derivative, $h_k = dh/df(\mathbf{x}_k)$, with respect to MLS values at nodes to a derivative with respect to MLS point values given the node coordinates, $\mathbf{x}_k$, and node derivatives, h_k:

$$G = \left\{ \frac{\sum_{k=1}^{k=N_n} h_k \exp\left(-\left(\frac{||\mathbf{x_k}-\boldsymbol{\xi}_j||}{r}\right)^2\right)}{\sum_{l=1}^{l=N_p} \exp\left(-\left(\frac{||\mathbf{x_k}-\boldsymbol{\xi}_l||}{r}\right)^2\right)}, \quad j = 1...N_p \right\} \tag{2.9}$$

The computational cost of this operation is $\mathcal{O}(N_n N_p^2)$. To minimize compute time for this operation, the loop on N_n is performed with a `Kokkos::parallel_reduce` which exploits GPU vectorization. One of the loops on N_p is limited to a local region to further reduce cost.

Multiple Program Multiple Data (MPMD) Computing Model

The objective gradients that are needed for topology optimization are computed using forward differencing,

$$\frac{dh}{dp_i} = \frac{h(p + \delta I_i) - h(p)}{\delta}, \tag{2.10}$$

where the vector I_i is one at index i and zero otherwise, and δ is a small number relative to the regularization length, α. The objective is evaluated $N_p + 1$ times for each gradient computation, and each objective evaluation involves at least one simulation of the relevant physics which will typically be computationally costly. To minimize run times, Plato Engine was used to perform sequential batches of simultaneous objective evaluations. In this section, a brief introduction to Plato Engine is given, and developments required to permit batched evaluation are described.

```
<Operation>
  <Name>Compute MLS Design</Name>
  <Function>ComputeMLSField</Function>
  <Input>
    <ArgumentName>MLS Values</ArgumentName>
  </Input>
  <MLSName>Design Geometry</MLSName>
</Operation>
```

Listing 2.1. Typical `Operation` specification.

```
<Performer>
  <Name>Alexa</Name>
  <PerformerID>1</PerformerID>
</Performer>
```

Listing 2.2. Typical `Performer` specification.

Coordination using Plato Engine

Plato Engine coordinates multiple application codes to perform a single task. In the context of topology optimization that task is to compute the design (topology) that minimizes some performance objective while satisfying state constraints, such as mechanical equilibrium, in the form of partial differential equations (PDEs). To permit the integration of multiple application codes for solving these PDE constraints, an abstraction called an `Application` is provided that allows the interaction between various service providers, or performers, to be defined at run time. Once a simulation code has been encapsulated in a concrete `Application` implementation, the basic steps for defining a task whether it's an optimization, parameter study, or coupled simulation, is as follows:

1. **Define Operations:** Each concrete implementation of an `Application` must also define the available `Operations` (e.g., Listing (2.1)). These definitions provide the name, input arguments and output arguments of any operation that the `Application` can provide during execution of the coordinated task.

2. **Define Performers:** An `Application` instance for which `Operations` have been defined and MPI ranks have been assigned is called a `Performer`. These `Performers` are used in the `SharedData` definitions below to specify how data flow during execution. A typical specification for a performer is shown in Listing (2.2). The performer ID must correspond to an ID given on the mpirun command line.

3. **Define SharedData:** Plato Engine manages the passage of data between `Performers`. The `SharedData` specification (see Listing (2.3) for an example) provides the name, data type, data owners, and data users among other information depending on the data layout. When the `SharedData` instance is an input/output to an `Operation`, the data

```
<SharedData>
  <Name>Optimization DOFs</Name>
  <Type>Scalar</Type>
  <Layout>Global</Layout>
  <Size>16</Size>
  <OwnerName>PlatoMain</OwnerName>
  <UserName>PlatoMain</UserName>
  <UserName>Alexa</UserName>
</SharedData>
```

Listing 2.3. Typical `SharedData` specification.

are transmitted from the owners to the users before/after the `Operation` is executed.

4. **Define Stages:** A `Stage` consists of input arguments, output arguments, and a collection of operations (see Listing (2.4) for an example). `Stage`s are available for execution through the `Plato::Interface` and can be used, for example, by an optimizer to collect objective and objective gradient information as needed.

A collection of example problems is available in the Plato Engine repository that demonstrates the basic usage. The following sections describe functionality that was added to Plato Engine as part of this work.

Syntax Expansion

A key aspect of the proposed approach is to evaluate as many of the perturbation calculations simultanenously as available hardware will permit. To enable many concurrent `Operations`, basic variable processing and control structure was added to Plato's input parsing. Listing (2.5) shows a typical set of variable definitions. These definitions can be placed in a single input file and imported as needed into the various configuration files using an `<include filename=defines.xml/>` statement. Once defined, variables are available for use in subsequent specifications using '{' and '}' to indicate an expression requiring evaluation.

As a practical mechanism for defining concurrent `Operations`, a `For`-loop construct was added to the Plato parsing stage. The syntax is shown in Listing (2.6) and consists of a dummy variable name and a previously defined `Array` variable. The contents of the `For` block are duplicated for each member of the given `Array` and any occurence of the dummy variable in the block is replaced. After processing, Listing (2.6) is expanded to the specification shown in Listing (2.7).

Variable definition and `For`-loop expansion provide a convenient method for defining concurrent `Operations` that arrange at run time into batches depending on available hardware. Listing (2.8) shows the basic structure for batched concurrent `Operations`. `Operations` that are specified within an `Operation` are executed concurrently, so the `Alexa_0`, `Alexa_1`, and

```
<Stage>
  <Name>Constraint</Name>
  <Input>
    <SharedDataName>Optimization DOFs</SharedDataName>
  </Input>
  <Operation>
    <PerformerName>Alexa</PerformerName>
    <Name>Compute MLS Design</Name>
    <Input>
      <ArgumentName>MLS Values</ArgumentName>
      <SharedDataName>Optimization DOFs</SharedDataName>
    </Input>
  </Operation>
  <Operation>
    <Name>Compute Constraint</Name>
    <PerformerName>Alexa</PerformerName>
    <Output>
      <ArgumentName>Constraint Value</ArgumentName>
      <SharedDataName>Constraint</SharedDataName>
    </Output>
  </Operation>
  <Output>
    <SharedDataName>Constraint</SharedDataName>
  </Output>
</Stage>
```

Listing 2.4. Typical `Stage` specification.

```
<Define name="N1"         type="int" value="3"                    />
<Define name="N2"         type="int" value="2"                    />
<Define name="Size1"                 value="5.0"                  />
<Define name="Size2"                 value="3.0"                  />
<Define name="Offset1"               value="{-Size1/2.0}"         />
<Define name="Offset2"               value="{-Size2/2.0}"         />
<Define name="Radius"                value="{Size1/N1}"           />
<Define name="NumPoints" type="int" value="{N1*N2}"               />
<Define name="NumComms"  type="int" value="{3}"                   />
<Define name="NumEvals"  type="int" value="{NumPoints/NumComms}"/>
<Define name="FD_Delta"              value="0.000001"             />
<Array  name="Evals"     type="int" from="0" to="{NumEvals-1}"   />
<Array  name="Comms"     type="int" from="0" to="{NumComms-1}"   />
```

Listing 2.5. Typical variable definitions.

```
<SharedData>
  <Name>Objective Gradient</Name>
  <Type>Scalar</Type>
  <Layout>Global</Layout>
  <Size>{N1*N2}</Size>
  <For var="I" in="Comms">
  <OwnerName>Alexa_{I}</OwnerName>
  </For>
  <UserName>PlatoMain</UserName>
</SharedData>
```

Listing 2.6. `SharedData` definition demonstrating use of expression evaluation and `For` block expansion.

```
<SharedData>
  <Name>Objective Gradient</Name>
  <Type>Scalar</Type>
  <Layout>Global</Layout>
  <Size>6</Size>
  <OwnerName>Alexa_0</OwnerName>
  <OwnerName>Alexa_1</OwnerName>
  <OwnerName>Alexa_2</OwnerName>
  <UserName>PlatoMain</UserName>
</SharedData>
```

Listing 2.7. `SharedData` definition after expansion.

```
<For var="J" in="Evals">
<Operation>
  <For var="I" in="Comms">
  <Operation>
    <PerformerName>Alexa_{I}</PerformerName>
    <Name>Compute Perturbed Design</Name>
    <Parameter>
      <ArgumentName>Perturbed Index</ArgumentName>
      <ArgumentValue>{I*NumEvals+J}</ArgumentValue>
    </Parameter>
  </Operation>
  </For>
</Operation>
</For>
```

Listing 2.8. Batched concurrent `Operations` specification.

```
<Performer>
  <For var="I" in="Comms">
  <Name>Alexa_{I}</Name>
  </For>
  <PerformerID>1</PerformerID>
</Performer>
```

Listing 2.9. Subdivided `Performer` specification.

`Alexa_2` performers will execute simultaneously with perturbed indices of 0, 2, and 4, respectively, then with 1, 3, and 5, respectively, in a second batch of simultaneous executions.

Subdividing Performers

To simplify the use of many performers, the `Performer` specification now accepts multiple names. The number of ranks assigned to the performer are divided evenly among the provided names to create multiple performers (see Listing (2.9)). The corresponding `mpirun` call is shown in Listing (2.10). In this example, each performer is given one rank, but this is not a limitation. Each created performer can be assigned multiple ranks for parallelism at the performer level.

Parameter Passing

A common data flow for solving topology optimization problems with Plato Engine is to broadcast the nodal design vector to the various performers that then compute the objective and/or objective gradient and transmit the data back to the optimizer. This approach has

```
mpirun \
-np 1 \
-x PLATO_PERFORMER_ID=0 \
-x PLATO_INTERFACE_FILE=interface.xml \
-x PLATO_APP_FILE=platoApp.xml \
PlatoMain platoMain.xml : \
-np 3 \
-x PLATO_PERFORMER_ID=1 \
-x PLATO_INTERFACE_FILE=interface.xml \
-x PLATO_APP_FILE=alexaApp.xml \
LGR_MPMD --input-config=mitchell_tri.xml
```

Listing 2.10. Example mpirun command with arguments.

been shown to scale well to many performers. However, the finite difference approach could have on the order of 10^3 simultaneous performers, so computing a nodal vector from the MLS points and broadcasting would create a severe communication bottleneck. To ensure good scaling to large problem sizes, the MLS points are broadcast to the performers, and the nodal design vector is computed locally. The performers are assigned perturbation evaluations by passing the indices as input parameters during execution. To use this approach, the MLS library was integrated with the performer to compute MLS fields locally, and parameter passing functionality was added to the Plato Engine. Listing (2.8) shows the syntax for passing a parameter to an operation.

Performer Integration

The original project plan was to integrate Sandia's Alegra code with the Plato Engine for testing of the MLS finite differencing approach. However, development of the MLS library and Plato Engine functionality took longer than anticipated, so an existing integration with the Sandia's Alexa code [2] was used to put the project back on schedule and ensure that the central goal of evaluating the MLS approach could be met.

Alexa is an adaptive multi-physics analysis code that has been designed to be portably performant on multiple architectures (GPU, CPU, many-core, etc.) and uses automatic differentiation to enable generation of gradient information. While the purpose of the current work is to develop and evaluate a non-invasive method for computing gradients, using an analysis code that can compute gradients allows for a direct comparison with the MLS approach. The results presented below were produced with Alexa's linear elastostatic physics to permit a direct comparison since the non-linear transient capability in Alexa cannot produce gradients.

Chapter 3

Results

In this chapter, the methods described previously are demonstrated on example applications in compliance minimization. The optimization problem is stated as follows:

$$\min_{\rho} \; h(u(\rho), \rho) \tag{3.1}$$

$$g(u(\rho), \rho) = 0 \tag{3.2}$$

$$\eta(\rho) \leq 0 \tag{3.3}$$

where h is a scalar valued objective, g is a vector valued function ($g = \{g_i, \; i...N_n\}$), η is a scalar valued linear constraint, u is a state vector ($u = \{u_i, \; i...N_n\}$), ρ is the nodal design vector ($\rho = \{\rho_i, \; i...N_n\}$), and N_n is the number of nodes in the finite element mesh. For linear compliance minimization,

$$h(u(\rho), \rho) = u(\rho)^T F \tag{3.4}$$

$$g(u(\rho), \rho) = K(\rho)u(\rho) - F \tag{3.5}$$

$$\eta(\rho) = vol(\rho) - V_o \tag{3.6}$$

$$K(\rho) = \mathop{\mathtt{A}}_{e} \int_{\Omega_e} B^T \left(CH(\rho)\right) B \; dV \tag{3.7}$$

$$F = \mathop{\mathtt{A}}_{e} \int_{\partial\Omega_e^t} N^T t \; dA \tag{3.8}$$

where $H(\rho)$ is the Heaviside projection defined above, B is the symmetric gradient operator, C is the material stiffness, and $\mathtt{A}$ denotes a finite element assembly operation. This problem statement is expressed in terms of the nodal design vector, ρ. If the minimization is to be performed relative to the MLS point values the problem statement is:

$$\min_{p} \; h(u(\rho(p)), \rho(p)) \tag{3.9}$$

etc., and the objective gradient can be computed either by finite differencing (Equation (2.10)) or by chain rule

$$\frac{dh}{dp} = \frac{dh}{d\rho}\frac{d\rho}{dp} \tag{3.10}$$

which is equivalent to Equation (2.9).

MLS versus Nodal Geometry

The MLS geometry representation uses substantially fewer degrees of freedom to define a model. In this section, a comparison is made between standard topology optimization using a nodal SIMP approach [4] and an MLS-based approach. The goal is to evaluate the ability of the MLS geometry representation to capture geometric detail relative to a nodal field, so the objective gradient information is computed directly (i.e., not using finite differencing) in the Alexa analysis code as a nodal field and then projected to a MLS field using the 'map derivative' functionality in the MLS library. This allows for a fairly direct evaluation of the effect of a lower resolution geometry description on the resulting geometry and optimal objective value. The input files for these problems are available in the Plato Engine repository in the `examples` directory in `Analyze_Mitchell2D_Heaviside` and `Analyze_Mitchell2D_MLS`.

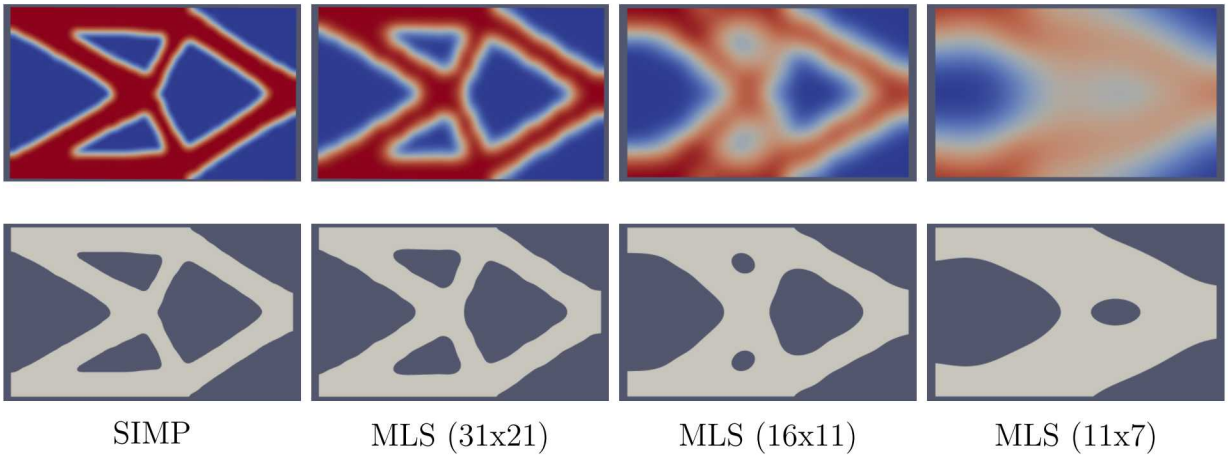


Figure 3.1. Comparison of 2D mitchell structures computed with Heaviside projection and MLS projection. The results generated with the Heaviside projection use 6852 optimization degrees-of-freedom (i.e., the number nodes in the mesh).

A comparison between SIMP and MLS results is shown in Figure (3.1). The SIMP design achieves a displacement of 1.023×10^{-3} using a nodal design vector with 6852 nodes. The MLS designs with resolutions (i.e., the number of points in X by Y directions) of 31x21, 16x11, and 11x7 achieve displacements of 1.142×10^{-3}, 1.724×10^{-3}, and 2.846×10^{-3}, which amounts to a loss in performance relative to the SIMP design of 11.6%, 68.5%, and 178%, respectively. These results show that the MLS representation can be used to reduce the dimensionality of the design (in this case, by 90%, 97%, and 99%, respectively) but at a cost in terms of the realized minimum objective. The rate of trade off between design resolution and performance is expected to vary case by case.

Projected versus Finite Difference Gradients

It was shown that the MLS representation can capture essential geometric detail with far fewer design degrees of freedom. In this section we demonstrate the use of a finite difference operation over this reduced set to generate the objective gradients necessary for topology optimization. This addresses the core question of this work: Can an existing analysis code that cannot produce gradient information be used for design optimization? The Alexa code does in fact have the ability to compute gradients, but for this comparison Alexa was only used to provide objective evaluations for the finite difference operation.

Figure (3.2) shows results for the 2d compliance minimization problem for various MLS resolutions. The input files for this problem are available in the Plato Engine repository in the `examples` directory in `Analyze_CompMin2D_Variables_MLS`. The optimized objective values are approximately the same as the mapped case above with the exception of the lowest resolution which has a substantially better stiffness in the finite difference case.

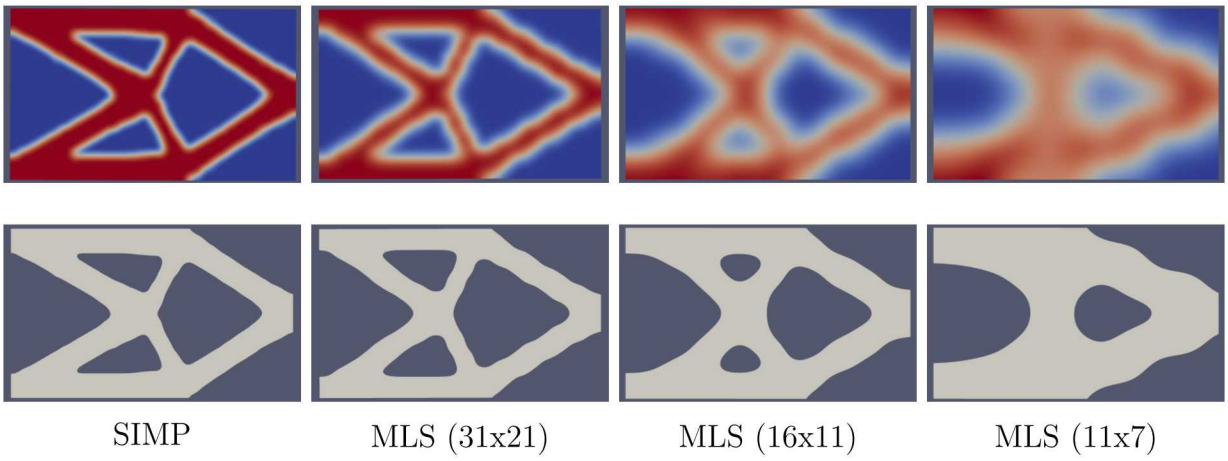


Figure 3.2. Comparison of 2D mitchell structures computed with Heaviside projection and MLS representation with finite differencing. The results generated with the Heaviside projection use 6852 optimization degrees-of-freedom (i.e., the number nodes in the mesh).

Scaling

Scaling performance was evaluated in the context of the 2d optimization problem above. Scaling data were generated on Ride which is an IBM Power8 platform with 16 CPU cores and four Nvidia Tesla P100 GPUs per node and is a testbed for anticipated HPC systems. The problem size (i.e., the number of nodes in the solution mesh and MLS points in the geometry description) was held fixed while the number of processors was varied (Figure (3.3)).

Ideal scaling is shown for reference. Scaling performance was not assessed beyond 128 CPU cores (32 GPUs) because platforms with sufficient hardware are not yet available, however, it is expected that this implementation will scale to much larger systems because the communication required between samples is minimal. Furthermore, the computational cost of an objective perturbation in this example is very low, so the communication overhead is amortized over a relatively low computational cost. More expensive objective evaluations will likely result in better scaling.

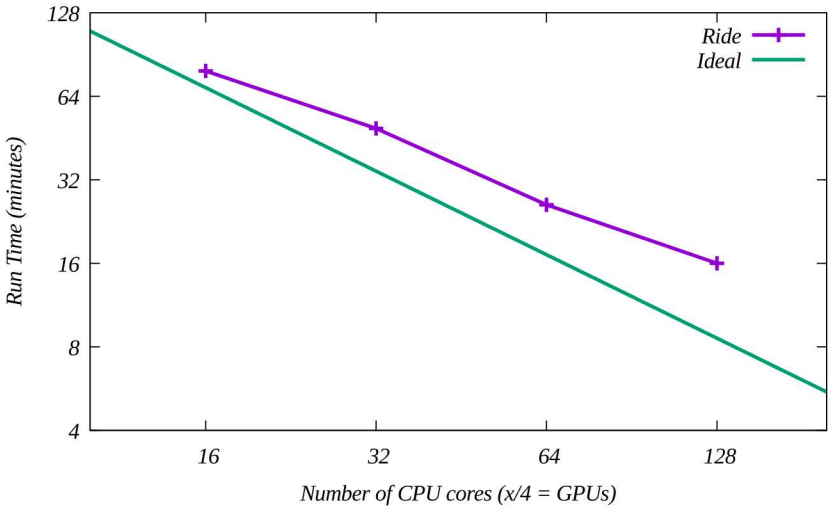


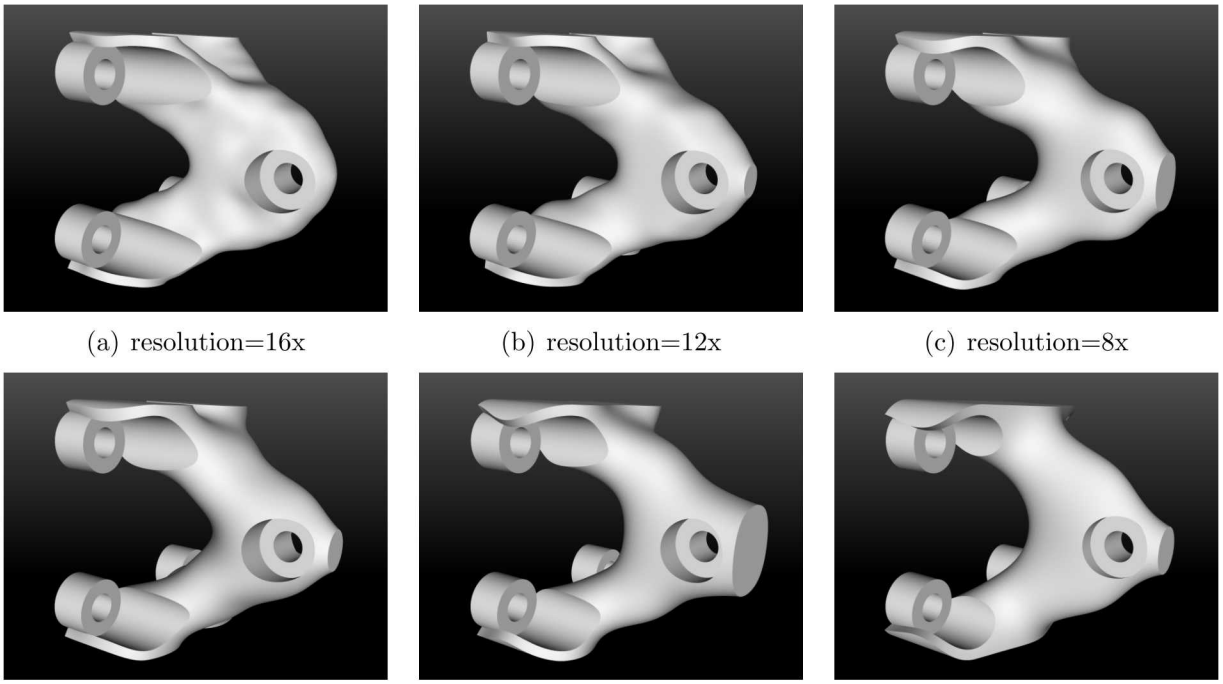
Figure 3.3. Scaling data for the 2D Mitchell problem on ride.sandia.gov (IBM Power8 CPUs, Nvidia Tesla P100 GPUs).

MLS Resolution

A 3d compliance minimization problem was solved at a range of MLS point resolutions to further evaluate the effect on resulting minimum objective value. The computed designs are shown in Figure (3.4), and a plot of the minimum objective value versus MLS resolution is shown in Figure (3.5). As in the 2d case, there is a clear trade-off between resolution of the MLS representation and the minimum achievable objective.

Smoothness of MLS Designs

The MLS function is infinitely differentiable (C^∞), so smooth designs are guaranteed at some length scale. However, to ensure smoothness at the scale of the part geometry, the interaction radius, r in Equation (2.2), must be chosen with some care. Figure (3.6) shows



(a) resolution=16x (b) resolution=12x (c) resolution=8x

(d) resolution=7x (e) resolution=6x (f) resolution=5x

Figure 3.4. Topologically optimized designs for minimum compliance. Effect of MLS resolution.

the effect on smoothness. At low radius values the design becomes voxelated as individual MLS points are visible, and at higher values the design is quite smooth on the scale of the part. Note that the flat surface on the right of the design is where the MLS geometry intersects the boundary of the design volume.

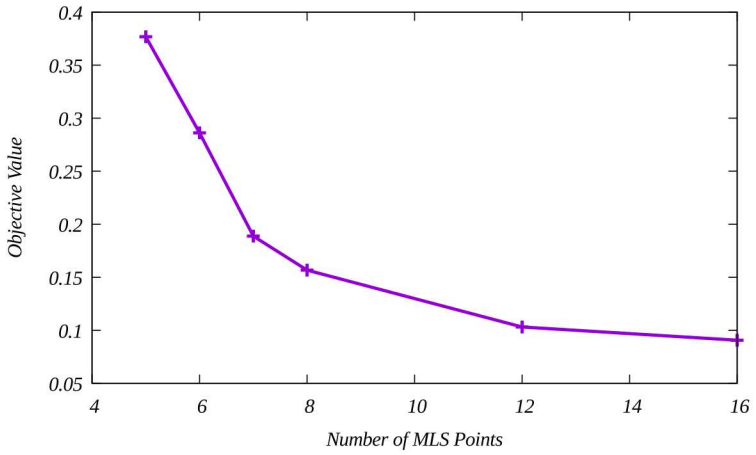
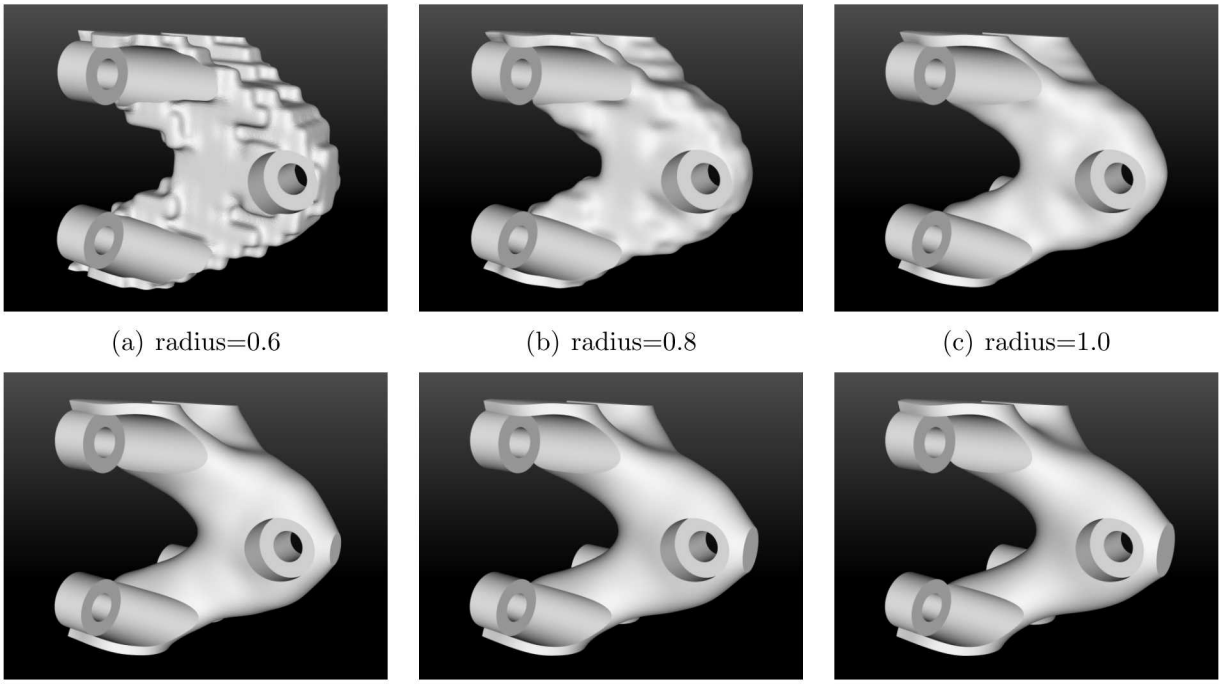


Figure 3.5. Effect of MLS resolution on minimum objective value for a 3d compliance minimization problem.



(a) radius=0.6 (b) radius=0.8 (c) radius=1.0

(d) radius=1.2 (e) radius=1.4 (f) radius=1.6

Figure 3.6. Topologically optimized designs for minimum compliance. Effect of interaction radius.

Chapter 4

Conclusions

This work produced a Moving Least Squares library in the Plato Engine tool set. In addition, functionality was added to the Plato Engine to allow finite differencing of performance objectives to generate the objective gradients necessary for topology optimization. The finite difference perturbations are computed in sequential batches depending on available hardware. The following conclusions can be drawn from this work:

1. The Moving Least Squares geometry representation can capture the essential features of a design with far fewer degrees of freedom (90 to 99 percent fewer is typical for the cases considered). This reduction comes at a cost in terms of the minimum acheivable objective value.

2. For the cases considered, gradients produced with finite differencing of the MLS geometry are effective for topology optimization and produce the same optimized design as a direct gradient method.

3. Parallel scaling of this approach and implementation is good on available hardware. This implementation is expected to scale well to larger systems, but these systems are not yet available.

This work has demonstrated the viablity of MLS with finite differencing for generating gradients from non-gradient application codes. Next steps in development include:

1. Evaluate the method with the non-linear, transient physics in the Alexa code. Since the Alexa code is already integrated with Plato Engine, the work required should be minimal.

2. Integrate Alegra with Plato Engine. As stated previously, the original project plan was to integrate Alegra to permit a rigorous evaluation of the method on a legacy code.

References

[1] T Belytschko, SP Xiao, and C Parimi. Topology optimization with implicit functions and regularization. *International Journal for Numerical Methods in Engineering*, 57(8):1177–1196, JUN 28 2003.

[2] Sandia National Laboratories. lgrtk, 2018. http://github.com/SNLComputation.

[3] Sandia National Laboratories. Plato engine, 2018. http://github.com/platoengine.

[4] O. Sigmund; K. Maute. Sensitivity filtering from a continuum mechanics perspective. *Struct Multidisc Optim*, 46:471–475, 2012.

DISTRIBUTION:

1 MS 0899 Technical Library, 9536 (electronic copy)