SANDIA REPORT
SAND2026-16145
Printed January 19, 2026

Sandia National Laboratories

FuSED – User's Manual – 5.28

FuSED Development Team:
Wilkins Aquino, Mark Chen, Andrew Kurzawski, Elizabeth R. Livingston, Cam McCormick, Justin Pepe, Clay Sanders, Chandler Smith, Ben Treweek, and Tim Walsh

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

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

Available to DOE and DOE contractors from

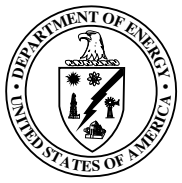
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

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

Available to the public from

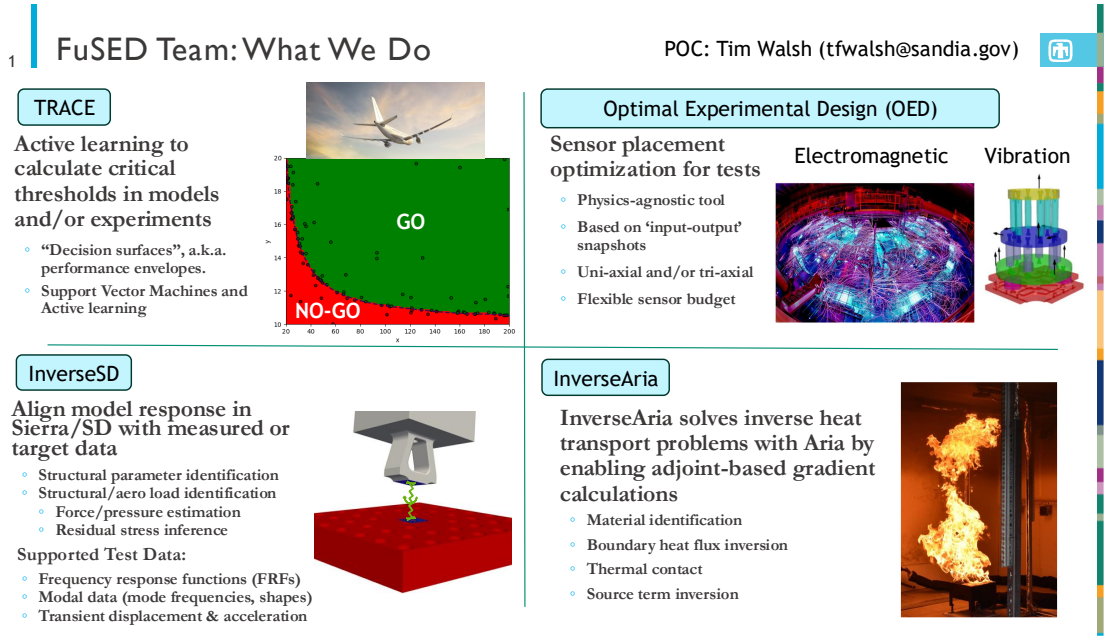
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods



ABSTRACT

The Fusion of Simulation, Experiment, and Data (FuSED) team provides a set of tools for solving inverse problems in structural dynamics (InverseSD) and thermal physics (InverseAria), a sensor placement optimization tool via Optimal Experimental Design (OED), and a decision boundary tool using SVMs (TRACE). These methods are used for designing experiments, model calibration, and verification/validation analysis of systems. This document provides a user's guide to the input for the four apps that are supported for these methods. Details of input specifications, output options, and optimization parameters are included.



Overview of the tools produced by the FuSED team.

Results from the use of this software should cite:
Wilkins Aquino, Mark J. Chen, Andrew J. Kurzawski, Elizabeth R. Livingston, Cameron A. McCormick, Justin Pepe, Clay M. Sanders, Chandler B. Smith, Benjamin C. Treweek, and Timothy F. Walsh. *FuSED – User's Manual – 5.28*. Sandia National Laboratories, 2026.

This page intentionally left blank.

CONTENTS

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

1. RELEASE NOTES

The sections in this chapter mainly describe new features, bug fixes, performance improvements, and features deprecated or removed in each new version of **FuSED**.

1.1. Release 5.28

New or Improved Features

New features:

- TRACE (TRACE Rapidly Acquires Contour Estimates)
  - Added the ability to handle continuous label data (instead of distinct class 0/1) with a user-defined threshold. See Section 5.3.3.
- InverseSD
  - Added the ability to save heterogeneous design variables to exodus output in the eigen-inverse material, spot weld, and damage identification cases at a user-defined interval. See Section 2.7.6.1
- OED
  - Enable triaxial sensor placement with gradient-based OED

Usability:

- InverseSD
  - The **OPTIMIZATION** block has been reworked for InverseSD. A blank optimization block is now valid syntax and will use ROL defaults. Advanced options now must be set via xml file. See Section 2.7.2.
- OED
  - Gradient-based OED includes baseline sensors in the final printed design.

Bug Fixes

- TRACE (TRACE Rapidly Acquires Contour Estimates)
  - User's models are now imported properly from a file. Previously, some users had to explicitly modify their `PYTHONPATH` environment variable, potentially compromising the security of their environment. See Section 5.2

1.2. Release 5.26

New or Improved Features

New features:

- TRACE (TRACE Rapidly Acquires Contour Estimates)
  - Full release in 5.26
  - Automatic hyperparameter tuning uses multi-dimensional bisection to choose the best settings for the user at each training iteration
- OED (Optimal Experimental Design)
  - Budget constraints can now be enforced with gradient-based optimization
  - Robust formulation for handling sensor drop-out/loss-of-data
- InverseAria
  - Support for sensors with arbitrary spatial locations
  - Support for adaptive time-stepping

Usability:

- InverseSD: Eigen MAC objective no longer supported; similar functionality can be achieved with the Matching objective with a scalar. Changes improved code robustness and usability.
- TRACE
  - Improved output options and re-factor of input deck
  - More frequent and more informative messages to users
  - Use of duck-typing to allow users more flexibility in defining their models
- OED
  - Input option XML checker
  - Expanded output information (objective function, sensors)

  - Training example on team wiki page for electromagnetics sensor placement optimization

Bug Fixes

- TRACE: Added saving of initial training data along with printouts for user debugging

1.3. Release 5.24

New or Improved Features

- TRACE (TRACE Rapidly Acquires Contour Estimates) improved post-processing enables more rapid construction of decision surfaces with uncertain model parameters.
- A new objective function enables model parameter estimation (stiffness, damping) to match SRS (Shock-Response Spectra) objective function in InverseSD.
- Improved Optimal Experimental Design (OED) code enhances runtime performance of gradient-based optimization algorithm for multi-observation (e.g. frequency domain) problems.
- InverseAria now supports hyper-reduction for adjoint solves when coupling with Pressio
- Training examples for using TRACE with Sierra/SM and Aria and video of quick-start tutorial added to team wiki page

Bug Fixes

- Allow consistent output between ROL and design variable output files for inverse methods

1.4. Release 5.22

New or Improved Features

- TRACE (TRACE Rapidly Acquires Contour Estimates) has been released as a beta capability for finding decision boundaries in high-dimensional parameter spaces.
- InverseSD supports interface identification through the inverse solution of distributed stiffness in spot weld elements in SD.
- The OED app now supports multi-sensor, multi-budget optimization of sensor placement.
- InverseAria now supports multiple design variable types, e.g. cases where it is desired to invert for flux and conductivity simultaneously

- In InverseSD, the MAC objective function for matching mode shapes and frequencies of SD models to test data has been improved with increased robustness of mode-matching and several bug fixes.
- InverseAria supports L2 regularization to mitigate effects of non-unique solutions for thermal inverse problems.

2. INVERSE METHODS IN SIERRA/SD

2.1. Inverse Solution Methods in Sierra/SD

Sierra/SD supports a wide variety of different analyses or solution methods. Input consists of an **Exodus** mesh file and a text input file. Solution methods are specified in the text input file in the solution section. For details on using **Sierra/SD**, including analysis types and solution methods not related to inverse problems, the reader is directed to the **Sierra/SD** User's Manual [21].

The **Solution** section of the input file defines the type of physics to simulate. Analysis types relevant to inverse problems are shown in Table 2-1.

Table 2-1. – Inverse Solution Types.

Solution Type	Description
eigen-inverse	Inverse solution to find material properties to produce given eigen solution
ModalFrf-inverse	Inverse solution to find load or power spectral density (PSD) to produce given modal frf
modaltransient-inverse	Inverse solution to find load to produce given modal transient
directfrf-inverse	Inverse solution to find load or material properties to produce given frequency response
transient-inverse	Inverse solution to find load or material properties to produce given transient solution

Each of the inverse solution methods described in Table 2-1 supports particular variables that can be inverted for in the solution process. Table 2-2 shows the matrix of supported inverse solution methods and the corresponding variables that can be extracted for each solution method. More details on the each inverse solution method and supported variables are given in the next sections.

Table 2-2. – Matrix of Supported Inverse Solution Types and Corresponding Design Variables.

Data Types	Material ID	Blk-Beta ID	Force ID	Interface ID	Objective Functions
FRF	X	X	X	X	L2 MECE (beta)
CPSD			X		L2
Modal	X			X	L2, MPE, MAC
transient	X	X	X		L2

2.2. DirectFRF-Inverse Solution Case

Parameter	Type	Default	Description

Table 2-3. – DirectFRF-Inverse Solution Case Parameters.

The `directfrf-inverse` solution method is used to solve an inverse problem for a direct frequency response analysis. As in a forward solution, most of the parameters of an inverse frequency response method are found in other sections[1]. The user provides complex displacements and/or pressures at a set of nodes in the model, and the solution to the inverse problem is a set of loads, materials, etc. that best correspond with the user's input.

The forward problem is defined in Eq. (2.2.1)

$$\left(\underbrace{K + i\omega C - \omega^2 M}_{\equiv A(\omega)} \right) \overline{u} = \overline{f}(\omega) \tag{2.2.1}$$

where $\overline{u}$ is the Fourier transform of the response u, and $\overline{f}$ is the Fourier transform of the applied force.The inverse equation is identical, but must be solved with optimization subject to regularization because measurements are available only at a subset of the analysis degrees of freedom.

The basic requirements for a `directfrf-inverse` simulation are as follows:

Optimization: Control over the optimization problem is specified in the **optimization** block. See Sec. 2.7.1 for further details.

Inverse-Problem: The **inverse-problem** block provides the connection to the measurement data. It is also where `design_variable` is specified (e.g., load, material, etc.).

[1]The forward solution supports a Padé expansion. This is not supported for inverse methods.

Truth Table: The truth table `data_truth_table` from the **inverse-problem** block is a list of the indices of the global node numbers (a.k.a. target nodes) where displacements or acoustic pressures are measured. See Sec. 2.7.4 for file format details.

Data File: Experimentally determined "target" displacements are read from `real_data_file` and `imaginary_data_file` specified in the **inverse-problem** block. See Sec. 2.7.4 for file format details.

Frequency: The frequencies at which the problem is solved are specified in the **frequency** block.

Sierra/SD uses the Rapid Optimization Library (ROL) for solving optimization problems. During the optimization solution ROL writes an output file, *ROL_Messages.txt* that contains convergence information. Section 2.7.9 contains a discussion of the output file that is written by ROL.

2.2.1. Load Identification

```
solution
  directfrf-inverse
end
inverse-problem
  design_variable = load
  data_truth_table = ttable.txt
  real_data_file = dataReal.txt
  imaginary_data_file = dataImag.txt
end
optimization
  xml_file = rolInput.xml
end
loads
  sideset 301
    inverse_load_type = spatially_constant
    pressure=10
    function = 1
end
```

Input 2.2.1. Direct frequency response load identification example input

Specifying `design_variable = load` applies inverse methods to determine sideset loads which best correspond with the measured displacements and/or acoustic pressures provided by the user. The material and model parameters do not change during the solution. For structures, the loads are pressures or tractions[2], and for acoustics, the loads are acoustic accelerations. Note that for structures, inversion is based on the signed magnitudes of the tractions; the direction of each traction is fixed.

[2]Moments and point forces are not currently supported.

An example input deck is given in input 2.2.1. In addition to the input blocks discussed in the beginning of this section, there are several others specific to `design_variable = load`:

Loads: See Sec. 2.7.7 for a description of the inverse parameters in the **loads** block for load identification problems.

Frequency: For unknown loads, the frequencies at which the problem is solved are independent. That is, a separate load identification is performed at each frequency.

Pressure or traction load identification through *modal* frequency response is also supported for structures (see Sec. 2.4.1). Because modal solutions are significantly cheaper than direct solutions, one approach might be to begin the inverse optimization with `modalfrf-inverse`, then use the output as the initial guess for a follow-up `directfrf-inverse` case. Section 2.7.8 contains a discussion of the current limitations with inverse load methods.

2.2.2. Material Identification

Specifying `design_variable = material` with the `directfrf-inverse` method applies inverse methods to determine material parameters when provided with both loads and structural displacements and/or acoustic pressures in a given finite element model[3]. The load parameters do not change during the solution. As in the previous section, the forward problem is defined in Eq. (2.2.1), and the inverse equation is identical but must be solved with optimization subject to regularization because measurements are available only at a subset of the analysis degrees of freedom. The solution provides the material parameters for elements in the model that are specified to have unknown materials.

In addition to the input blocks discussed at the beginning of this section, there are several others specific to `design_variable = material`:

Block: See Sec. 2.7.5 for a description of the **block** specifications for material inverse problems.

Material: See Sec. 2.7.6 for a description of the provides **material** specifications for material inverse problems.

Frequency: For unknown materials, the same set of material properties apply for every frequency in the simulation, except in the case of frequency-dependent material properties.

Viscoelastic material identification is also supported using measured homogenized complex bulk and shear moduli. This capability is limited to structural-only problems where all material blocks are isotropic viscoelastic.

[3] As the system matrices (and consequently the modes) change at every inverse iteration, `design_variable` cannot be set to `material` for `modalfrf-inverse` problems.

2.2.3. Multi-Experiment Material Identification

In the same manner as `design_variable = material` described in the previous section, `design_variable = multi_material` may be used to apply inverse methods in the frequency domain to determine material parameters. Here, multiple inverse problems are combined. For instance, if two different load and displacement conditions result in two separate responses for the same set of material properties, this method will use both responses to determine a single set of material properties.

Most parameters for a multi-experiment inverse frequency method are similar to those for a single-experiment inverse frequency method. The differences occur in the following sections:

Loads: The **loads** block must be empty for this solution case; anything the user specifies here will be overwritten by what they specify in the **load** block.

Load: Each **load** block provides a separate set of loads for each experiment individually.

Inverse-Problem: In addition to the parameters discussed earlier, the **inverse-problem** section must include values for `nresponses` for the number of experiments and `loadID` to specify a list of loads, one for each experiment.

2.2.4. Circuit Parameter Identification For Piezoelectric Modeling

In piezoelectric modeling with electric circuits, the circuit parameters are real constants, and can be any combination of resistance, capacitance and inductance values. Specifiying `inverse_material_type = homogeneous` in a circuit block input can be used to identify these constants. This capability is currently only supported for the `directfrf-inverse` solution case. User must also specify upper and lower bounds for each circuit parameter used in a given circuit block. For example, input 2.2.2 inverts for three circuit parameters defined in Block 1. The keyword `inverse_material_type = homogenous` declares that circuit parameters in this block are treated as inverse parameters. The upper and lower bounds for each parameters are specificed with keywords `capacitance_bounds`, `resisance_bounds` and `inductance_bounds`. The upper and lower constants are user specified real values.

If needed, user can also identify circuit parameters concurrently with material model as decribed in 2.2.2.

```
BLOCK 1
 electrical_circuit
 inverse_material_type = homogeneous
 capacitance = 1e-9
 resistance = 50
 inductance = 1e-6
 capacitance_bounds = 1e-12 1e-6
 resistance_bounds = 1 100
 inductance_bounds = 1e-9 1e-3
```

```
END
```

Input 2.2.2. Directfrf circuit parameter identification example input

2.3. Eigen-Inverse Solution Case

Parameter	Type	Default	Description
nmodes	*Integer*	10	Number of modes to extract.
shift	*Real*	-1.0e6	Shift to apply to matrix system to allow solving singular systems.
untilfreq	*Real*	Inf	Target frequency to reach.
ModalFilter	*string*	none	Modal filter to define modes to retain.
modalAdjoint-Solver	*gdsw*| *camp*| *both*	camp	Select solver for the inverse problem (eigenvector material identification only)
repeatEigenval-ueTolerance	*Real*	1.e-4	Tolerance for repeated eigenvalues.

Table 2-4. – Eigen-Inverse Solution Case Parameters.

The `eigen-inverse` solution method is used to solve an inverse problem for an eigen analysis. In this solution method, only material identification is currently supported. Specifying `design_variable = material` applies inverse methods in the modal domain to determine material properties on a block or element when provided with modal frequencies and mode shapes. The user specifies some of the lowest modes of the structure, and optionaly the mode shapes of the structure at locations in the model.

The standard parameters for modal analysis also apply here. The analysis requires input both for measurement data and for control of various optimization parameters. See the following sections for details:

Optimization: Control over the optimization problem is specified in the **optimization** block. See Sec. 2.7.1 for further details.

Inverse-Problem: The **inverse-problem** block provides the connection to the measurement data. Of particular importance are the parameters `modal_data_file` and `modal_weight_table`, which are described further in Sec. 2.7.4. Also necessary is the parameter `design_variable = material`.

Block: For material ID problems, the optimization strategy for 3D element blocks is specified using the **inverse_material_type** keyword within the **block** section. This is also the section in which optimization parameters are specified for `joint2g` elements. See Sec. 2.7.5 for details.

Material: For material ID problems, additional options to control the identification of 3D elements are specified in the **material** section. These include which material parameters are being inverted for and the bound constraints on those parameters. For details, see Sec. 2.7.6.

Sierra/SD uses the Rapid Optimization Library (ROL) for solving optimization problems. During the optimization solution ROL writes an output file, *ROL_Messages.txt* that contains convergence information. Section 2.7.9 contains a discussion of the output file that is written by ROL.

2.3.1. Eigenvalue Material Identification

```
solution
  eigen-inverse
    nmodes=12
    shift=-1e5
end
inverse-problem
  design_variable =  material
  modal_data_file =  modal_data.txt
  modal_truth_table =  modal_truth.txt
end
optimization
  xml_file = rolInput.xml
  scaleDesignVars = yes
end
```

Input 2.3.1. Eigen material identification example input

In the case of eigenvalue optimization, only the modal frequencies are included in the objective function. An example input is shown in input 2.3.1. The theory for this problem is available in [5]. The objective function for the eigen value problem is given as:

$$J(\boldsymbol{\lambda}, \boldsymbol{u}, \boldsymbol{p}) = \sum_{i=1}^{N} \left[\frac{\beta_i}{2} \left(\frac{\lambda_i - \lambda_{mi}}{\lambda_{mi}} \right)^2 \right] + \mathcal{R}(\boldsymbol{p}). \tag{2.3.1}$$

Where m is the number of modes, β_i is zero or one, λ_i is the computed eigenvalue, λ_{mi} is the measured eigenvalue, and $\mathcal{R}(\boldsymbol{p})$ is the regularization term.

2.3.2. Eigenvector Material Identification

In this case, both the eigenvalues (modal frequencies) and eigenvectors (mode shapes) are matched in the inverse solution. A detailed description of the theory and implementation details of this solution case is given in [5]. To use this capability, it is necessary to specify the keywords `data_file` and `data_truth_table` for the eigenvector data and eigenvector truth table, respectively, which specify the modal shape amplitude data and truth table information. The latter allows one to differentiate between a tri-axial and uni-axial accelerometer.

The objective function extends Eq. (2.3.1) by adding the eigenvector term.

$$J(\boldsymbol{\lambda}, \boldsymbol{u}, \boldsymbol{p}) = \sum_{i=1}^{N} \left[\frac{\beta_i}{2} \left(\frac{\lambda_i - \lambda_{mi}}{\lambda_{mi}} \right)^2 + \frac{\kappa_i}{2} \frac{\|\boldsymbol{u}_i - \boldsymbol{u}_{mi}\|_{\boldsymbol{Q}}^2}{\|\boldsymbol{u}_{mi}\|_{\boldsymbol{Q}}^2} \right] + \mathcal{R}(\boldsymbol{p}) \,, \tag{2.3.2}$$

where $\boldsymbol{u}_i$ is the computed eigenvector, $\boldsymbol{u}_{mi}$ is the measured eigenvector, and $\boldsymbol{Q}$ is the observation matrix obtained from the truth table.

The issues an implementation must handle include repeated modes, crossing modes, the singular adjoint linear system, and eigenvector scaling.

2.3.2.1. Repeated Modes

When the computed or measured data contain repeated eigenvalues, the associated eigenvectors are not deterministic and steps are taken to orthogonalize these modes with respect to the measured data. The measured data must be sufficient enough for this orthogonalization for the inverse problem to converge.

Modes with repeated roots commonly occur in structures exhibiting geometric symmetry (e.g. a beam with a symmetric cross section about two axes). A collection of measurement locations that is not geometrically symmetric may introduce minor numerical error into the orthogonalization operation, as the geometric symmetry of the rotated repeated modes will not be preserved. When repeated modes are expected in a structure, specifying geometrically-symmetric measurement locations can improve the unique rotation of computed mode shapes into the direction of measured shapes. Furthermore, repeated modes can also occur at iterations in the inverse process even when no repeated modes are present in the measured data. In the SOLUTION section, the user can set `repeatEigenvalueTolerance` (default value is $1e-4$) as the threshold value to decide whether two computed eigenvalues are repeated.

When a pair of repeated modes is detected, such that the $|\lambda_i - \lambda_{i+1}| < \epsilon_{tol}$ (where ϵ_{tol} is the repeated root tolerance), we employ a rotation and mass-reorthonormalization strategy to create consistent, unique pairings from the pair of computed modes to a pair of measured repeated modes.

Let $\{\boldsymbol{u}_i, \boldsymbol{u}_j\}$ represent two mode shapes with repeated roots; $\{\boldsymbol{v}_i, \boldsymbol{v}_j\}$ are a pair of measured mode shapes targeted for the rotation operation. (Note, the indices i, j of target measured shapes do not necessarily correspond to the indices of the computed repeated modes.) We form linear

combinations of the computed modes as their projected components in the direction of the measured mode shapes:

$$\tilde{\boldsymbol{u}}_i = \frac{\boldsymbol{u}_i^T[Q]\boldsymbol{v}_i}{\boldsymbol{v}_i^T[Q]\boldsymbol{v}_i}\boldsymbol{u}_i + \frac{\boldsymbol{u}_j^T[Q]\boldsymbol{v}_i}{\boldsymbol{v}_i^T[Q]\boldsymbol{v}_i}\boldsymbol{u}_j \tag{2.3.3}$$

$$\tilde{\boldsymbol{u}}_j = \frac{\boldsymbol{u}_i^T[Q]\boldsymbol{v}_j}{\boldsymbol{v}_j^T[Q]\boldsymbol{v}_j}\boldsymbol{u}_i + \frac{\boldsymbol{u}_j^T[Q]\boldsymbol{v}_j}{\boldsymbol{v}_j^T[Q]\boldsymbol{v}_j}\boldsymbol{u}_j \tag{2.3.4}$$

where $[Q]$ is the observation matrix. We then follow by mass orthonormalizing the two rotated, computed eigenvectors. We first scale $\tilde{\boldsymbol{u}}_i$ by $\alpha_i = \tilde{\boldsymbol{u}}_i^T[M]\tilde{\boldsymbol{u}}_i$, the mass inner product for $\tilde{\boldsymbol{u}}_i$:

$$\bar{\boldsymbol{u}}_i = \frac{1}{\alpha_i}\tilde{\boldsymbol{u}}_i \tag{2.3.5}$$

We then form $\bar{\boldsymbol{u}}_j$ as:

$$\hat{\boldsymbol{u}}_j = \tilde{\boldsymbol{u}}_j - \left(\bar{\boldsymbol{u}}_i^T[M]\tilde{\boldsymbol{u}}_j\right)\bar{\boldsymbol{u}}_i \tag{2.3.6}$$

$$\bar{\boldsymbol{u}}_j = \frac{1}{\alpha_j}\hat{\boldsymbol{u}}_j \tag{2.3.7}$$

where $\alpha_j = \hat{\boldsymbol{u}}_j^T[M]\hat{\boldsymbol{u}}_j$.

We note that the resulting $\{\bar{\boldsymbol{u}}_i, \bar{\boldsymbol{u}}_j\}$ remain mass-orthonormal,

$$\bar{\boldsymbol{u}}_i^T[M]\bar{\boldsymbol{u}}_j = \delta_{ij}. \tag{2.3.8}$$

2.3.2.2. Rigid Body Modes

Eigen-inverse solution method uses modal test data to invert for material properties in structures. As such, it does not match any model information to test data for the rigid body modes of the system. Thus, we recommend using the `num_rigid_mode` parameter in the **parameters** section to specify how many rigid modes are expected to be in the model and test data to avoid rigid body modes being matched in the optimization process. We also recommend using the truth table to exclude rigid body modes from being matched to test data.

```
PARAMETERS
 num_rigid_modes 6
END
```

Input 2.3.2. Rigid Body Modes example input

2.3.2.3. Mode Swapping/Crossing

If the computed eigenvalues of a structure are ordered from smallest to largest, the ordering of mode shapes will typically change as the material parameters are varied. This also causes non-differentiability in the objective function, which causes difficulties in gradient-based optimization. Mode tracking refers to maintaining a correspondence of eigenpairs (eigenvalue and eigenvectors) between an original and an updated system throughout changes in the eigenproblem. Measured data is often incomplete, having only a few measured data points (physical accelerometer locations) on a model with millions of degrees of freedom. An incorrect mode swap results in a discontinuity in the slope of the objective function.

A mode tracking algorithm is used to minimize eigenvector misfit at each optimization step.

2.3.2.4. Singular Solve

The Adjoint Solution is singular due to the fact that the eigenvector u_i is in the kernel of the coefficient matrix. In order for a solution to exist, the right hand side must be orthogonal to u_i . Additionally, if rigid body modes ($\lambda = 0$) or repeated mode are present, components of the corresponding eigenvectors must also be removed from the right hand side before the solve. Even when this is done, however, the resulting system of equations is singular and a Helmholtz (indefinite) problem, which presents significant computational cost and robustness challenges for iterative linear solvers.

The `modalAdjointSolver = camp` (default for eigenvector inversion) option enables a new solver that uses a modal superposition of the previously computed eigenvectors to solve this system of equations. When using the `camp` solver, it is recommended to request more modes than contained in the measured data, and use the truth table file to remove these modes from the optimization part of the solution.

2.3.2.5. Computed Eigenvector Scaling

An important consideration in eigenvector optimization is that the mode shapes computed in Sierra/SD are by default *mass normalized*. Measured modal shape amplitudes, on the other hand, could present with very different scalings, since any eigenvector can be scaled by an arbitrary scale factor and will still be a valid eigenvector. Thus, the eigen-inverse solution method includes an automatic re-scaling of the computed mode shapes in the optimization so that they have the same norm as the measured mode shapes. This re-normalization allows them to be properly differenced in the objective function. We note that this internal re-scaling requires no user intervention.

If the norms of the measured eigenvectors differ substantially from the norms of the eigenvectors computed in Sierra/SD, then the re-scaling described in the previous paragraph is necessary to correctly determine the next iteration of the design variables. The scaled computed eigenvector $\tilde{\boldsymbol{u}}_i$ can be written as

$$\tilde{\boldsymbol{u}}_i = \alpha_i \boldsymbol{u}_i \,, \tag{2.3.9}$$

where $\alpha_i = |\boldsymbol{u}_{mi}|/|\boldsymbol{u}_i|$ such that the norm of $\tilde{\boldsymbol{u}}_i$ is identical to the norm of eigenvector $\boldsymbol{u}_{mi}$. With this change, the eigenvector term in Eq. (2.3.2) becomes

$$J(\boldsymbol{u}) = \sum_{i=1}^{N} \frac{\kappa_i}{2} \frac{\|\alpha \boldsymbol{u}_i - \boldsymbol{u}_{mi}\|_{\boldsymbol{Q}}^2}{\|\boldsymbol{u}_{mi}\|_{\boldsymbol{Q}}^2}, \tag{2.3.10}$$

A corresponding change to the gradient $J_{\boldsymbol{u}}$ is also required, but this change is not discussed here.

2.3.2.6. Eigen Objective

Instead of the default matching objective in Eq. (2.3.2), the user can specify a modal projection error (MPE) objective by setting the **eigen_objective** parameter to `mpe` in the **inverse-problem** section. The two options are a least squares MPE objective, specified by setting the **mpe_algorithm** to `ls` (along with setting `eigen_objective = mpe`), and a singular value decomposition (SVD) based MPE objective, specified by setting the **mpe_alorithm** parameter to `svd`.

2.3.2.7. Projection Mode Selection

By default, the modes used in the MPE objective are selected according to the indices of their respective eigenvalues, using the N modes with the lowest eigenvalues. If the user wishes to select the N modes with the greatest modal assurance criterion (MAC) values, they can set the **projection_mode_selection** parameter to `mac` instead of `lowest`.

2.4. ModalFRF-Inverse Solution Case

Parameter	Type	Default	Description
nmodes	*Integer*	10	Number of modes to extract.
shift	*Real*	-1.0e6	Shift to apply to matrix system to allow solving singular systems.
untilfreq	*Real*	Inf	Target frequency to reach.
ModalFilter	*string*	none	Modal filter to define modes to retain.
lfcutoff	*Real*	-Inf	Exclude any modes below this frequency from the modal computation. Often used to exclude rigid body modes.

Table 2-5. – ModalFRF-Inverse Solution Case Parameters.

The `modalfrf-inverse` solution method is used to solve an inverse problem for a modal frequency response analysis. The modal FRF method is similar to the direct FRF method, except the user must specify the number of modes `nmodes`. As in a forward solution, most of the parameters in an inverse modal frequency response analysis are found in other sections, and as in a `directfrf-inverse` problem, the user provides complex displacements and/or acoustic pressures at a set of nodes in the model.

The forward problem is defined in equation (2.2.1). The inverse equation is identical, but must be solved with optimization subject to regularization because measurements are available only at a subset of the analysis degrees of freedom.

The basic requirements for a `modalfrf-inverse` simulation are as follows:

Optimization: Control over the optimization problem is specified in the **optimization** block. See Sec. 2.7.1 for further details.

Inverse-Problem: The **inverse-problem** block provides the connection to the measurement data. It is also where `design_variable` is specified (e.g., load, material, etc.).

Truth Table: The truth table `data_truth_table` from the **inverse-problem** block is a list of the indices of the global node numbers (a.k.a. target nodes) where displacements or acoustic pressures are measured. See Sec. 2.7.4 for file format details.

Data File: Experimentally determined "target" displacements are read from `real_data_file` and `imaginary_data_file` specified in the **inverse-problem** block. See Sec. 2.7.4 for file format details.

Frequency: The frequencies at which the problem is solved are specified in the **frequency** block.

Sierra/SD uses the Rapid Optimization Library (ROL) for solving optimization problems. During the optimization solution ROL writes an output file, *ROL_Messages.txt* that contains convergence information. Section 2.7.9 contains a discussion of the output file that is written by ROL.

2.4.1. Load Identification

```
solution
  modalfrf-inverse
  nmodes 100
end
inverse-problem
  design_variable = load
  data_truth_table = ttable.txt
  real_data_file = data.txt
  imaginary_data_file = data_im.txt
end
optimization
  xml_file = rolInput.xml
end
```

```
loads
  sideset 6
    inverse_load_type = spatially_constant
    pressure=1
    function = 1
  sideset 6
    inverse_load_type = spatially_constant
    ipressure=1
    function = 2
end
function 1
  type linear
  data 1 3
  data 2 4
end
function 2
  type linear
  data 1 5
  data 2 6
end
```

Input 2.4.1. Modal frequency response load identification example input

Specifying `design_variable = load` applies inverse methods to determine sideset loads which best correspond with the measured displacements and/or acoustic pressures provided by the user. The material and model parameters do not change during the solution. For structures, the loads are pressures or tractions[4], and for acoustics, the loads are acoustic accelerations. Note that for structures, inversion is based on the signed magnitudes of the tractions; the direction of each traction is fixed.

An example input deck is given in input 2.4.1. In addition to the input blocks discussed in the beginning of this section, there are several others specific to `design_variable = load`:

Loads: See Sec. 2.7.7 for a description of the inverse parameters in the **loads** block for load identification problems.

Frequency: For unknown loads, the frequencies at which the problem is solved are independent. That is, a separate load identification is performed at each frequency.

Section 2.7.8 contains a discussion of the current limitations with inverse load methods.

[4]Moments and point forces are not currently supported.

2.4.2. PSD Load Identification

```
solution
  modalfrf-inverse
  nmodes 100
end
inverse-problem
  design_variable = psd_load
  data_truth_table = ttable.txt
  data_file = data.txt
end
optimization
  xml_file = rolInput.xml
end
loads
  sideset 6
    inverse_load_type = spatially_constant
    pressure=1
    function = 1
  sideset 6
    inverse_load_type = spatially_constant
    ipressure=1
    function = 2
end
function 1
  type linear
  data 1 3
  data 2 4
end
function 2
  type linear
  data 1 5
  data 2 6
end
```

Input 2.4.2. Modal frequency response PSD load identification example input

Specifying `design_variable = psd_load` applies inverse methods to determine a *load PSD* (power spectral density) as an output when provided with the PSD of acoustic pressures or structural displacements and a finite element model. The input is similar to modal FRF load identification, with a key exception. The design variables must be defined such that there are independent real and imaginary parts of the force, traction, or pressure. Thus, there should be twice the number of design variables as the dimension of load PSD, corresponding to real and imaginary parts of the load. These design variables must be entered into the input file in the order

of the load index. Furthermore, for each load index, the design variable for the real part should be immediately followed by the design variable for the imaginary part. It is also important to note that this example is not demonstrating a random load. That is shown in the next section.

An example input deck is given in input 2.4.2. In addition to the input blocks discussed in the beginning of this section, there are several others specific to `design_variable = psd_load`:

Data File: Experimentally determined "target" response PSDs are read from the `psd_data_file` described in Sec. 2.7.4.

Loads: See Sec. 2.7.7 for a description of the inverse parameters in the **loads** block for load identification problems. **loads** block.

Frequency: For unknown loads, the frequencies at which the problem is solved are independent. That is, a separate load identification is performed at each frequency.

2.4.3. Random PSD Load Identification

```
solution
  modalfrf-inverse
  nmodes 100
end
inverse-problem
  design_variable = psd_load
  data_truth_table = ttable.txt
  psd_data_token = inputCPSD
  data_type = accel
end
optimization
  xml_file = rolInput.xml
end
LOADS
   nodeset 117
      force 1 0 0
      inverse_load_type SPATIALLY_CONSTANT
      function approx_zero1
   nodeset 117
      iforce 1 0 0
      inverse_load_type SPATIALLY_CONSTANT
      function approx_zero4
   nodeset 219
      force 0 1 0
      inverse_load_type SPATIALLY_CONSTANT
      function approx_zero2
   nodeset 219
      iforce 0 1 0
```

```
      inverse_load_type SPATIALLY_CONSTANT
      function approx_zero5
   nodeset 353
      force 0 0 1
      inverse_load_type SPATIALLY_CONSTANT
      function approx_zero3
   nodeset 353
      iforce 0 0 1
      inverse_load_type SPATIALLY_CONSTANT
      function approx_zero6
END
FUNCTION approx_zero1
   type linear
   data 0.0  1e-6
   data 1e10 1e-6
END
FUNCTION approx_zero2
   type linear
   data 0.0  1e-6
   data 1e10 1e-6
END
FUNCTION approx_zero3
   type linear
   data 0.0  1e-6
   data 1e10 1e-6
END
FUNCTION approx_zero4
   type linear
   data 0.0  1e-6
   data 1e10 1e-6
END
FUNCTION approx_zero5
   type linear
   data 0.0  1e-6
   data 1e10 1e-6
END
FUNCTION approx_zero6
   type linear
   data 0.0  1e-6
   data 1e10 1e-6
END
```

Input 2.4.3. Modal frequency response PSD random load identification example input

Specifying `design_variable = psd_load` applies inverse methods to determine a *load PSD*

(power spectral density) as an output when provided with the PSD of acoustic pressures, structural accelerations (like in this example) or structural displacements and a finite element model. The input is similar to modal FRF load identification, with two exceptions. First, the design variables must be defined as a cpsd matrix with real and imaginary parts representing the force, traction, or pressue. Thus, there should be twice the number of design variables as the dimension of load CPSD matrix, corresponding to real and imaginary parts of the load. These design variables must be entered into the input file in the order of the load index. Furthermore, for each load index, the design variable for the real part should be immediately followed by the design variable for the imaginary part. Second, a different data file format is required, although the truth table format is identical. See Sec. 2.7.4 for further details.

An example input deck is given in input 2.4.3. In addition to the input blocks discussed in the beginning of this section, there are several others specific to `design_variable = psd_load`:

Data File: Experimentally determined "target" response PSDs are read from a file with a name identical to the `psd_data_token` with "Response.txt" appended to the end described in Sec. 2.7.4.

Loads: See Sec. 2.7.7 for a description of the inverse parameters in the **loads** block for load identification problems. **loads** block.

Frequency: For unknown loads, the frequencies at which the problem is solved are independent. That is, a separate load identification is performed at each frequency.

2.5. ModalTransient-Inverse Solution Case

Parameter	Type	Default	Description
nmodes	*Integer*	10	Number of modes to extract.
shift	*Real*	-1.0e6	Shift to apply to matrix system to allow solving singular systems.
untilfreq	*Real*	Inf	Target frequency to reach.
ModalFilter	*string*	none	Modal filter to define modes to retain.
lfcutoff	*Real*	-Inf	Exclude any modes below this frequency from the modal computation. Often used to exclude rigid body modes.
time_step	*Real*		Time step size.
nsteps	*Integer*		Number of time steps to take.
start_time	*Real*	0.0	Solution case start time.
nskip	*Integer*	1	Results output frequency.
rho	*Real*	1	Select time integrator.
load	*Integer*		Load to apply during solution case.
write_files	*all*| *none*| *output*| *history*	all	Controls which result files are written during this solution.

Table 2-6. – ModalTransient-Inverse Solution Case Parameters.

```
solution
  modaltransient-inverse
  nsteps = 100
  time_step = 1e-3
  nskip = 1
  nmodes = 100
end
inverse-problem
  design_variable = load
  data_truth_table = ttable.txt
  data_file = dataReal.txt
  tikhonovParameter 1.0e-5
end
optimization
```

```
  xml_file = rolInput.xml
end
loads
  sideset 301
    inverse_load_type = spatially_constant
    pressure=10
    function = 1
end
```

Input 2.5.1. Modal transient load identification example input

The `modaltransient-inverse` solution method is used to solve an inverse problem for a modal transient analysis. In this solution method, only load identification is supported. Specifying `design_variable = load` applies inverse methods to determine sideset loads with best correspond with measured displacements and/or acoustic pressures provided by the user in the modal time domain. This capability differs from load identification in a `transient-inverse` problem (Sec. 2.6.1.1) only in that modal superposition is used to reduce computation time. As with forward analysis, the modaltransient solution will converge to the direct solution as the number of modes increases. See the SierraSD verification manual for an example of this convergence[22].

The parameters for load identification in a direct `transient-inverse` problem also apply in the `modaltransient-inverse` case. The latter also requires the parameter `nmodes`, the number of eigen modes calculated in the forward solve, as well as any additional parameters needed for the eigen solution case. The eigen modes need only be calculated once, and then can be re-used for each inverse iteration. Note that the Tikhonov parameter can be used to mollify instability in the early time history.

An example is shown in input 2.5.1. The following input blocks are needed for `modaltransient-inverse` with `design_variable = load`:

Optimization: Control over the optimization problem is specified in the **optimization** block. See Sec. 2.7.1 for further details.

Inverse-Problem: The **inverse-problem** block provides the connection to the measurement data. It is also where `design_variable = load` must be specified.

Truth Table: The truth table (`data_truth_table` from the **inverse-problem** block is a list of the indices of the global node numbers (a.k.a. target nodes) where displacements or acoustic pressures are measured. See Sec. 2.7.4 for file format details.

Data File: Experimentally determined "target" displacements are read from `data_file` specified in the **inverse-problem** block. See Sec. 2.7.4 for file format details.

Loads: See Sec. 2.7.7 for a description of the inverse parameters in the **loads** block for load identification problems.

Section 2.7.8 contains a discussion of the current limitations with inverse load methods.

Sierra/SD uses the Rapid Optimization Library (ROL) for solving optimization problems. During the optimization solution ROL writes an output file, *ROL_Messages.txt* that contains convergence information. Section 2.7.9 contains a discussion of the output file that is written by ROL.

2.6. Transient-Inverse Solution Case

Parameter	Type	Default	Description
time_step	*Real*		Time step size.
nsteps	*Integer*		Number of time steps to take.
start_time	*Real*	0.0	Solution case start time.
nskip	*Integer*	1	Results output frequency.
rho	*Real*	1	Select time integrator.
load	*Integer*		Load to apply during solution case.
write_files	*all*| *none*| *output*| *history*	all	Controls which result files are written during this solution.

Table 2-7. – Transient-Inverse Solution Case Parameters.

The `transient-inverse` solution method is used to solve in inverse problem for a time domain analysis. With a few exceptions, the parameters for the forward `transient` solution case apply to this solution method as well. The user provides a time series of displacements and/or pressures at a set of nodes in the model, and the solution to the inverse problem is a set of loads, materials, etc. that best correspond with the user's input.

The basic requirements for a `transient-inverse` simulation are as follows:

Optimization: Control over the optimization problem is specified in the **optimization** block. See Sec. 2.7.1 for further details.

Inverse-Problem: The **inverse-problem** block provides the connection to the measurement data. It is also where `design_variable` is specified (e.g., load, material, etc.).

Truth Table: The `data_truth_table` from the **inverse-problem** block is a list of the indices of the global node numbers (a.k.a. target nodes) where displacements or acoustic pressures are measured. See Sec. 2.7.4 for file format details.

Data File: Experimentally determined "target" displacements are read from `data_file` specified in the **inverse-problem** block. See Sec. 2.7.4 for file format details.

Sierra/SD uses the Rapid Optimization Library (ROL) for solving optimization problems. During the optimization solution ROL writes an output file, *ROL_Messages.txt* that contains convergence information. Section 2.7.9 contains a discussion of the output file that is written by ROL.

2.6.1. Design Variables

2.6.1.1. Load Identification

Specifying `design_variable = load` applies inverse methods to determine sideset loads which best correspond with the measured displacements and/or acoustic pressures provided by the user. The material and model parameters do not change during the solution. For structures, the loads are pressures or tractions[5], and for acoustics, the loads are acoustic accelerations. Note that for structures, inversion is based on the signed magnitudes of the tractions; the direction of each traction is fixed.

```
solution
  transient-inverse
  nsteps = 100
  time_step = 1e-3
  nskip = 1
end
inverse-problem
  design_variable = load
  data_truth_table = ttable.txt
  data_file = dataReal.txt
end
optimization
  xml_file = rolInput.xml
end
loads
  sideset 301
    inverse_load_type = spatially_constant
    pressure=10
    function = 1
end
```

Input 2.6.1. Transient Load Identification Example

An example input deck is given in input 2.6.1. In addition to the input blocks discussed in the beginning of this section, there is another that is specific to `design_variable = load`:

Loads: See Sec. 2.7.7 for a description of the inverse parameters in the **loads** block for load identification problems.

[5]Moments and point forces are not currently supported.

Section 2.7.8 contains a discussion of the current limitations with inverse load methods.

2.6.1.2. Material Identification

Specifying `design_variable = material` with the `transient-inverse` method applies inverse methods to determine material parameters when provided with both loads and structural displacements and/or acoustic pressures in a given finite element model[6] The load parameters do not change during the solution, which provides the material parameters for elements in the model that are specified to have unknown materials.

In addition to the input blocks discussed in the beginning of this section, there are several others specific to `design_variable = material`:

Block: See Sec. 2.7.5 for a description of the **block** specifications for material inverse problems.

Material: See Sec. 2.7.6 for a description of the provides **material** specifications for material inverse problems.

2.6.2. Objective Functions

The objective function used in the transient problem is selected in the `INVERSE-PROBLEM` block using the `transient_objective` keyword.

2.6.2.1. Tracking Objective Function

This is the default objective function if the `transient_objective` keyword is not defined. All design variables (i.e. `load`,`material`,`shape`) will work with the default tracking objective.

2.6.2.2. SRS Objective Function

Instead of using time history data, a user may specify the shock response spectra (SRS) across a range of frequencies. This is still contained within the `transient-objective` because the full time history data is used in evaluating the SRS reponse. The objective function is selected by applying `srs` for the `transient_objective` keyword in the `INVERSE-PROBLEM` block.

```
solution
  transient-inverse
  nsteps = 100
  time_step = 1e-3
  p_norm = 10
end
```

[6] As the system matrices (and consequently the modes) change at every inverse iteration, `design_variable` cannot be set to **material** for `modaltransient-inverse` problems.

```
inverse-problem
  design_variable = material
  transient_objective = srs
  data_type accel
  data_truth_table = ttable.txt
  data_file = data.txt
end
frequency
  freq_min 300.
  freq_max 800.
  freq_step 50.
  nodeset 1
  acceleration
end
optimization
  scaleDesignVars = on
  xml_file = rolInput.xml
end
```

Input 2.6.2. Transient SRS Material Identification Example

An portion of an example input deck is given in input 2.6.2. In order to formulate the inverse problem using SRS data, the maximum is replaced with a p-norm, which must be specified in the `solution` block. The implementation of SRS also relies on acceleration data, so the `data_type` listed in the `inverse-problem` block must be `accel`. Finally, although this is a transient solution case, a `frequency` block must be specified, listing the range of frequencies used for evaluating the SRS.

2.7. Inverse Options in Sierra/SD

Inverse problems optimize parameters to reproduce experimental results. Inverse methods include transient and direct frequency response load identification, direct frequency response material identification, and material identification from eigenvalues. The methods are based on solving optimization problems, with the goal to minimize the norm of the difference between measured and predicted data. More detail is provided in the references found in the theory notes. Inverse methods for identifying an unknown material (2.3, 2.2.2) or an unknown load (2.2.1, 2.6.1.1) require solution block input. There may be additional input required, such as the specification of test data results 2.7.4 and the specification of the parameters in the Optimization 2.7.1 and Inverse Problem 2.7.3 sections. Input 2.7.1 illustrates a *partial* input for a "directfrf-materialid" problem. Highlighted portions of the input are outlined below.

Sierra/SD uses the Rapid Optimization Library (ROL) as an optimization engine. Portions of the ROL documentation can be found on the Trilinos website.[7]

```
solution
  directfrf-inverse
end
optimization
  xml_file = rolInput.xml
end
inverse-problem
  design_variable = material
  data_truth_table = ttable.txt
  real_data_file =  data.txt
  imaginary_data_file =  data_im.txt
end
block 1
  inverse_material_type=homogeneous
  material 1
end
block 2
  inverse_material_type=known
  material 2
end
material 1
  isotropic
    density 10
    G 1
    K 1
end
material 2
  isotropic
    density 1
    G 2
    K 2
end
```

Input 2.7.1. Sample "directfrf-inverse" input for material identification. Portions of the input that are specific to inverse methods are emphasized.

[7] https://trilinos.org/packages/rol

2.7.1. Optimization

The **optimization** section provides options to control the optimization strategy as part of an inverse method such as material identification. Parameters for the optimization section are listed in Table 2-8, and an example is shown in input 2.7.2.

Sierra/SD uses the Rapid Optimization Library (ROL) [12], which is a Trilinos [9] package for large-scale optimization. ROL is particularly well suited for the solution of optimal design, optimal control and inverse problems in large-scale engineering applications. The currently supported methods are trust region and line search, and the corresponding parameters for these methods are listed in Section 2.7.2. These
parameters are only a subset of the parameters available in ROL. Example XML files can be found at `/projects/sierra/toolset/*version*/contrib/FuSED_tools/xml_templates/sd_inverse`

We note that for *source inversion* problems that involve **directfrf-inverse** , **transient-inverse** or **modaltransient-inverse** solution cases, we recommend using Krylov-based methods such as *line search* with the *newton-krylov* option, or *trust region* with *truncatedcg* option.

In the case of *material inversion* problems, the best algorithm is problem-dependent, and may require some experimentation to arrive at the optimal parameters. We note that for material inverse problems, the parameter **scaleDesignVars** has been shown to significantly help convergence. Accordingly, this parameter defaults to `yes`, but can be set to `no` to facilitate convergence.

Sierra/SD currently uses ROL methods for unconstrained and bound-constrained optimization with line searches and trust regions To provide the context for the parameter section, we review some notation and provide references to optimization textbooks.

Suppose $\mathcal{X}$ is a Hilbert space of functions mapping Ξ to $\mathbb{R}$. For example, $\Xi \subset \mathbb{R}^n$ and $\mathcal{X} = L^2(\Xi)$ or $\Xi = \{1, \ldots, n\}$ and $\mathcal{X} = \mathbb{R}^n$. We assume that the objective function $f : \mathcal{X} \to \mathbb{R}$ is twice continuously Fréchet differentiable and that the bound constraints $a,\ b \in \mathcal{X}$ are given with $a \leq b$ almost everywhere in Ξ. We focus on methods for solving unconstrained and bound-constrained optimization problems,

$$\underset{x}{\text{minimize}} \quad f(x) \qquad \text{and} \qquad \underset{x}{\text{minimize}} \quad f(x) \quad \text{subject to} \quad a \leq x \leq b\,,$$

respectively. The methods implemented in ROL utilize derivative information and two strategies for guaranteed (global) convergence from remote starting points, line searches and trust regions. We use the notation $\nabla f(x)$ to the note the gradient of f at x and $\nabla^2 f(x)$ to denote the Hessian of f at x.

Line-search methods. Let x_k be the k-th optimization iterate. For unconstrained problems, line search methods compute an update to x_k in the form of

$$x_{k+1} = x_k + \alpha_k s_k\,,$$

where s_k is a descent vector, and $\alpha_k > 0$ is a scalar. The vector s_k can be computed using a variety of methods, including steepest descent, nonlinear conjugate gradients, quasi-Newton (secant) methods and Newton-Krylov methods, see [18, Ch. 3, 5, 8 and 6, resp.] and [8, Ch. 6.3]. Section

2.7.2 lists the parameters corresponding to the choice of the method to compute the descent vector s_k. To compute the scalar α_k, a line search approximately minimizes the one-dimensional function $\phi_k(\alpha) := f(x_k + \alpha s_k)$, i.e., it approximately solves the optimization problem

$$\underset{\alpha}{\text{minimize}} \quad \phi_k(\alpha) := f(x_k + \alpha s_k)\,.$$

In general, the approximate minimizer α_k must satisfy sufficient decrease and curvature conditions to guarantee global convergence [18, Ch. 3]. **Sierra/SD** uses the *cubic interpolation* line search from ROL, which includes a backtracking procedure that satisfies the Armijo sufficient decrease condition

$$\phi_k(\alpha_k) \leq \phi_k(0) + c_1\alpha_k\phi_k'(0) \qquad \Longleftrightarrow \qquad f(x_k + \alpha_k s_k) \leq f(x_k) + c_1\alpha_k\langle \nabla f(x_k), s_k\rangle_{\mathcal{X}}\,,$$

where $0 < c_1 < 1$, and does not require a curvature condition. An initial guess for the line-search parameter can be specified if steepest descent or nonlinear conjugate gradient methods are used for the computation of the descent vector s_k. For bound constrained problems, the line search is a *projected* search. That is, the line search approximately minimizes the one-dimensional objective function

$$\phi_k(\alpha) = f(P_{[a,b]}(x_k + \alpha s_k))\,,$$

where $P_{[a,b]}$ denotes the projection onto the upper and lower bounds. Such line-search algorithms result in projected gradient, projected quasi-Newton and projected Newton algorithms (see below).

Trust-region methods. For unconstrained problems, given the k-th iterate x_k trust-region methods compute the trial step s_k by approximately solving the trust-region subproblem

$$\underset{s}{\text{minimize}} \quad \frac{1}{2}\langle B_k s, s\rangle_{\mathcal{X}} + \langle g_k, s\rangle_{\mathcal{X}} \qquad \text{subject to} \qquad \|s\|_{\mathcal{X}} \leq \Delta_k\,,$$

where $B_k \in L(\mathcal{X}, \mathcal{X})$ is an approximation of $\nabla^2 f(x_k)$, g_k approximates $\nabla f(x_k)$, and $\Delta_k > 0$ is the trust-region radius. The approximate minimizer s_k must satisfy the fraction of Cauchy decrease condition

$$-\frac{1}{2}\langle B_k s, s\rangle_{\mathcal{X}} - \langle g_k, s\rangle_{\mathcal{X}} \geq \kappa_0 \|g_k\|_{\mathcal{X}} \min\left\{\Delta_k,\ \frac{\|g_k\|_{\mathcal{X}}}{1 + \|B_k\|_{L(\mathcal{X},\mathcal{X})}}\right\}$$

for some $\kappa_0 > 0$ independent of k. ROL implements several trust-region methods, including Truncated CG (conjugate grdient), Cauchy Point, Dogleg, and Double Dogleg. See [18, Ch. 4] and [8, Ch. 6.4] for more info. Section 2.7.2 lists the parameters (Subproblem Solver) corresponding to the choice of the method. Additionally, the user can specify the initial trust-region radius, Δ_0 (Initial Radius, Section 2.7.2). For bound constrained problems, ROL employs projected gradient, projected secant, and projected Newton-type methods. These methods prune variables based on the binding set (see below) and run standard trust-region subproblem solvers on the remaining variables. To ensure sufficient decrease, ROL then performs a modified projected line search.

Bound Constraints. The bound constraint methods in ROL require the active set of an iterate x_k,

$$\mathcal{A}_k = \{\, \xi \in \Xi \,:\, x_k(\xi) = a(\xi) \,\} \cap \{\, \xi \in \Xi \,:\, x_k(\xi) = b(\xi) \,\}\,.$$

The active set is the subset of Ξ corresponding to points in which x_k is equal to the upper or lower bound. The complement of the active set (called the inactive set), $\mathcal{I}_k = \mathcal{A}_k^c = \Xi \setminus \mathcal{A}_k$, is the subset of Ξ corresponding to points in which x_k is strictly between a and b. Given $\mathcal{A}_k$ and the gradient $g_k = \nabla J(x_k)$, we define the binding set as

$$\mathcal{B}_k = \{\, \xi \in \Xi \,:\, x_k(\xi) = a(\xi),\ -g_k(\xi) < 0 \,\} \cap \{\, \xi \in \Xi \,:\, x_k(\xi) = b(\xi),\ -g_k(\xi) > 0 \,\}\,.$$

The binding set contains the values of $\xi \in \Xi$ such that if $x_k(\xi)$ is equal to either the upper and lower bound, then $(x_k - g_k)(\xi)$ will violate bound. For both projected line-search and trust-region methods, the step is computed by fixing the variables in the active set and only optimizing over the inactive variables. That is,

$$s_k|_{\mathcal{A}_k} = 0 \quad \text{and} \quad s_k|_{\mathcal{I}_k} \text{ are free.}$$

Considering active, inactive and binding variables results in poor overall performance. To circumvent this behavior, ROL employs ϵ variations of this set. Namely, if $\epsilon > 0$, then

$$\mathcal{A}_k^\epsilon = \{\, \xi \in \Xi \,:\, x_k(\xi) \le a(\xi) + \epsilon \,\} \cap \{\, \xi \in \Xi \,:\, x_k(\xi) \ge b(\xi) - \epsilon \,\}$$
$$\mathcal{B}_k^\epsilon = \{\, \xi \in \Xi \,:\, x_k(\xi) \le a(\xi) + \epsilon,\ -g_k(\xi) < 0 \,\} \cap \{\, \xi \in \Xi \,:\, x_k(\xi) \ge b(\xi) - \epsilon,\ -g_k(\xi) > 0 \,\}\,.$$

The ϵ inactive set is similarly defined as the complement of the ϵ active set. ROL dynamically controls ϵ so that as an algorithm approaches the optimal solution, ϵ decreases to zero. [10, 4, 15, 6, 7].

Krylov methods. Both line-search and trust-region methods may involve iterative methods of the Krylov type in the step computation. If such methods are requested, the stopping conditions for these sub-solvers can be defined through the parameters Absolute_Krylov_tol, Relative_Krylov_tol and Max_iter_Krylov described in Section 2.7.2. Scenarios requiring Krylov methods are triggered, for instance, if TruncatedCG is selected for a trust-region algorithm or if newtonkrylov is selected for a line-search algorithm.

2.7.2. Optimization Options in Sierra/SD

2.7.2.1. Minimum Parameters

In the optimization block, most parameters will be set in the XML file. There are two infrequently used parameters that can be set directly in the input deck these are as follows:

Parameter	type	default	Description
boundConstraints	Yes/No	Yes	Bound constraints on design variables
scaleDesignVars	Yes/No	Yes	Controls scaling of design variables by initial guess to make them nondimensional

Table 2-8. – Optimization Section Parameters

An example of this block is shown below.

```
OPTIMIZATION
    boundConstraints = yes //default and optional
    scaleDesignVars = yes //default and optional
    xml_file = rolInput.xml //optional
END
```

Input 2.7.2. Optimization Section Example

If a user simply desires to use default ROL parameters, no xml file needs to be included. This means that an empty **OPTIMIZATION** block is valid input if all defaults are desired.

2.7.2.2. Optional Parameters

In the **OPTIMIZATION** block, there are some additional rarely used parameters that may be set. They are as follows:

Parameter	Type	Default	Description
scalingType	Diagonal/Uniform	Diagonal	Type of design variable scaling
scaleDesignWeight	real	1	Scales the design variables by a user-specified constant
constraint_type	Volume / Max_Stress / Surface_Curvature	N/A	Type of constraint
volume_fraction_constraint	real	N/A	Fraction for volume constraint

Table 2-9. – Rarely Used Optimization Section Parameters

XML files have parameter lists and parameters. The syntax for a parameter list and parameter are given below.

```
〈 ParameterList name="List Name"/〉
  〈 Parameter name="name" type="string/int/bool/double" value="value" 〉
〈/ ParameterList〉
```

For the remainder of the document, we will remove the angular brackets and the type for convenience and space.

All XML input files must have a **ParameterList name = "Inputs"** at the beginning that encompasses all the parameters the user wishes to pass to the algorithm. This is actually the minimum XML file that can be used to run InverseSD. Any additional parameters are optional and

only need to be added if the user wants to set them to something besides the deafults. Running with the following xml file is equivalent to not including one in the input deck.

```
ParameterList name = "Inputs"
ParameterList
```

A number of tem-
plate xml files using the options discussed in this section and more can be found in the toolset repo at `/projects/sierra/toolset/*version*/contrib/FuSED_tools/xml_templates/sd_inverse`.

The **Status Test** sublist is used to control high level optimization parameters such as the optimality criteria and number of iterations. Some of the most commonly used parameters are shown below.

```
ParameterList name="Status Test"
    Parameter name="Gradient Tolerance" type="double" value="1.e-10"
    Parameter name="Constraint Tolerance" type="double" value="1.e-8"
    Parameter name="Objective Tolerance" type="double" value="1.e-20"
    Parameter name="Step Tolerance" type="double" value="1.e-12"
    Parameter name="Iteration Limit" type="int" value="20"
ParameterList
```

By default, a **Trust Region** step type is used. In some cases, a user may want to declare additional parameters related to the **Trust Region** step. If that is desirable, the following may be declared in the XML file:

```
ParameterList name="Step"
    ParameterList name="Trust Region"
        Parameter name="Subproblem Solver" type="string"
                                    value="Truncated CG"
        Parameter name="Initial Radius" type="double" value="-1"
        Parameter name="Maximum Radius" type="double" value="1e8"
    ParameterList
ParameterList
```

The **Subproblem Solver** can be set to **Truncated CG**, **Cauchy Point**, **Dogleg**, and **Double Dogleg**. The **Initial Radius** is another optional parameter that can be set if desirable.

At times, a user may want to tune additional parameters in the **General** parameters. This includes enabling the **"Use as Hessian"** option and the **Krylov** sublist there parameters. Some of the most commonly used parameters are shown below:

```
ParameterList name="General"
    ParameterList name="Krylov"
        Parameter name="Absolute Tolerance" type="double" value="1.e-6"
        Parameter name="Relative Tolerance" type="double" value="1.e-3"
        Parameter name="Iteration Limit" type="int" value="20"
```

```
    ParameterList
    ParameterList name="Secant"
        Parameter name="Use as Hessian" type="bool" value="true"
    ParameterList
ParameterList
```

There are other top level parameters that would be defined directly under **Inputs**. Commonly used parameters, including diagnostic parameters, are shown below:

```
Parameter name="Use Reduced SimOpt" type="bool"  value="false"
Parameter name="Inexact Hessian-Times-A-Vector" type="bool" value="false"

Parameter name="checkReducedGradient" type="bool" value="false"
Parameter name="checkReducedHessian" type="bool" value="false"
Parameter name="checkHessianSymmetry" type="bool" value="false"
Parameter name="finiteDifferenceScale" type="bool" value="false"
```

Finite difference options are also defined directly under **Inputs**, but also in the **Composite Step** sublist. Commonly used parameters are shown below:

```
Parameter name="Use FD Hessian" type="bool"  value="false"
Parameter name="Use FD Hessian SimOpt" type="bool" value="false"
ParameterList name="Composite Step"
    ParameterList name="Optimality System Solver"
        Parameter name="Fix Tolerance" type="bool" value="true"
        Parameter name="Nominal Relative Tolerance" type="double"
                                                    value="1e-8"
    ParameterList
    ParameterList name="Tangential Subproblem Solver"
        Parameter name="Iteration Limit" type="int" value="20"
        Parameter name="Relative Tolerance" type="double" value="1e-4"
    ParameterList
ParameterList
```

While the **Trust Region** step type is recommended, an **Augmented Lagrangian** or **Line Search** step type may be used. The following may be declared in the XML file to use **Line Search**:

```
ParameterList name="Step"
    ParameterList name="Line Search"
        Parameter name="User Defined Initial Step Size" type="bool"
                                                        value="true"
        Parameter name="Initial Step Size" type="double" value="1.0"
        Parameter name="Type" type="string" value="Line Search"
        ParameterList name="Descent Method"
            Parameter name="Type" type="string" value="Newton-Krylov"
        ParameterList
```

```
        ParameterList name="Curvature Condition"
            Parameter name="Type" type="string" value="Null Curvature
                                                       Condition"
        ParameterList
        ParameterList name="Line-Search Method">
            Parameter name="Type" type="string" value="Cubic
                                              Interpolation"
        ParameterList
    ParameterList
ParameterList
```

The **Descent Method** can be set to **Nonlinear CG**, **Steepest Descent**, **Quasi-Newton Method** (often referred to as secant method), **Newton-Krylov**, or **Newton's Method** (requires useTransferMatrix=on 2-10). The **Curvature Condition** is optional, but, if desired may be set to **Null Curvature Condition** as well as other advanced options which all have additional, optional parameters. The **Line-Search Method** and **Initial Step Size** parameters are also optional.

ROL offers many more advanced options and suboptions for various purposes. While having many knobs can be helpful, it is unecessary for most InverseSD problems. Submit a support ticket for additional guidance on advanced options.

2.7.3. Inverse-Problem

The **inverse-problem** block of the input deck connects externally defined data describing the test. The format for the files is described in the Inverse Data Files section (2.7.4). This section also controls parameters of the optimization (such as regularization). Options for the **inverse-problem** section are included in Table 2-10.

Inverse Problem Parameters	**Load Id**		**Material Id**	
	Transient	**FRF**	**FRF**	**Eigen**
General Parameters				
design_variable	R	R	R	R
data_file	R	NA	NA	NA
data_truth_table	R	R	R	NA
data_weight_table	NA	NA	NA	-
real_data_file	NA	R*	R	NA
imaginary_data_file	NA	R*	R	NA
psd_data_token	NA	R*	R	NA
modal_data_file	NA	NA	NA	R
modal_weight_table	NA	NA	NA	R
data_type	-	-	-	-
useTransferMatrix	NA	-	-	NA
link_blocks	NA	NA	-	-
Regularization Parameters				
gradientSurfaceRegParameter	-	-	NA	NA
tikhonovParameter	-	-	-	-
gradientTikhonovRegularizationParameter	NA	NA	-	-
MECE_penalty	NA	NA	-	NA
Multi-Experiment Parameters				
nresponses	NA	NA	NA	NA
loadID	NA	NA	-	NA

Table 2-10. – Inverse-Problem parameters. Items marked with "-" are optional. Items marked 'R' are required, and items marked NA are not applicable. An asterisk is present for FRF-Load Id because either the real and imaginary data parameters or the psd data token are required, depending on if we are solving for a random signal.

2.7.3.1. Regularization Parameters

gradientSurfaceRegParameter a penalty term, penalizing jumps in load input parameters between neighboring patches. Increasing the regularization parameter decreases checker-boarding on multi-patch outputs.

tikhonovParameter a penalty term penalizing large values of the design variables. A larger term forces smaller design variables.

MECE_penalty a penalty term, penalizes the size of the misfit error between the measured and predicted data. Unlike least squares methods, MECE methods to not strive to exactly reproduce the measured data. Increasing the penalty term decreases the misfit error.

2.7.3.2. Multi-Experiment Parameters

nresponses the number of experients in a multi-experiment inverse problem. If not specified by the user, this parameter defaults to 1. This parameter must match the number of data files listed for 'real_data_file' and 'imaginary_data_file'.

loadID list of load identifiers, each corresponding to a separate experiment in a multi-experiment inverse problem. The length of this list must match the value specified for 'nresponses'.

2.7.3.3. Transfer Matrix Option

The transfer matrix is a linear map between the design space and the state space for linear inverse problems such as source inversion. When the **useTransferMatrix** is set to true, the transfer matrix is internally computed and stored, and explicitly used to compute the gradient and the full hessian matrix. This option may signficantly decrease computational time at the expense of in core memory for select optimization problems with some or all of the following characteristics: 1. small number of design variables, 2. linear inverse problems with uncertainty, and 3. PSD source inversion problems (future release capability). In addition to increased computational performance, this option will allow the user to solve linear inverse problems using the full newton method (see Table 2-8).

Currently, the useTransferMatrix is only applicable to solution type **inverse-directfrf** problems, and must be flagged with `beta` when executing **Sierra/SD**.

2.7.3.4. Link Blocks Option

The **link_blocks** keyword enables connection of two inverse material blocks to the same, shared set of unknown material properties. By default, referencing the same material from multiple unknown blocks will otherwise cause a fatal error in **Sierra/SD**. Linking blocks reduces the number of design variables in the inverse problem and should result in faster convergence than if the linked blocks arrived at the same set of material properties independently.

Note, the 'link_blocks' option is only applicable to material identification problems and is limited to connecting two blocks with shared, homogeneous material properties. Additionally, only one 'link_blocks' command is currently permitted. Usage of 'link_blocks' is indicated by including **link_blocks <integer> <integer>** in the **inverse-problem** section, where the two integer arguments are the linked block ID numbers. For example, with **link_blocks 1 2**, both **block 1** and **block 2** will share the same unknown material properties.

2.7.4. Inverse Data Files

The interface for measured data involves several data files. The specific files needed depends on the solution method, as summarized in Table 2-10. This section describes the data files and data files formats. Input 2.7.3 provides an example of a frequency domain inverse problem.

```
inverse-problem
  design_variable = load | material
  data_truth_table = truthTable.txt
  real_data_file = dataReal.txt
  imaginary_data_file = dataImag.txt
  data_type = disp | accel | moduli | voltage
end
```

Input 2.7.3. The inverse problem section for direct frequency response sets the names of the files that contain user specified data.

In the case of a structural-only problem, the `data_type` can be set to either `disp` or `accel`, indicating that the incoming data is in the form of displacements or accelerations, respectively. In addition to displacement and acceleration data, the `data_type` can also be set to `moduli` for viscoelastic material identification problems. For acoustics-only problems, `data_type` is not applicable as the incoming data always corresponds to acoustic pressure. For models incorporating piezoelectric materials, setting `data_type` to `voltage` indicates that the experimental data is in the form of voltages. The parameter `data_type` defaults to `disp` if not specified.

Data Truth Table

The `data_truth_table` file contains the **global** node numbers (a.k.a. target nodes) where the experimental data measurements are given. The first line in the file contains the number of points where measurements are given, and the remaining lines contain the global node numbers where the experimental data is specified. If `data_type` is set to `moduli`, the `data_truth_table` file contains the node numbers where the displacements are computed. **IMPORTANT:** If superelements are used, the data truth table must not include superelement internal degrees of freedom.

Figure 2-1 provides a simple example of the truth table format for an acoustics-only problem, Figure 2-2 provides an example for a structural-only problem, and Figure 2-3 provides an example for a structural acoustics problem. Note that for acoustics-only, the file consists of the list of nodes where the acoustic measurements were taken. For voltage based problem, the file looks identical to an acoustic-only problem, where the listed nodes represent the locations where voltage measurements were taken. For structural-only problems, each line in the truth table contains 4 columns that indicate the node numbers where measurements were taken, and then 3 columns of binary (0/1) input that indicate which dofs (i.e. x, y, and z) are active in the optimization. For structural acoustics problems, Fig. 2-3 shows that each line contains 5 columns, with the first column again indicating the node number, and the remaining four columns contain either 1 or 0, which allows to turn on/off the x, y, z, and $pressure$ degrees of freedom in the optimization.

The data truth table identifies the location of measurement data. For example, if the experimental data is collected at three microphones, which correspond to nodes 10, 120, and 3004, then the `data_truth_table` file is as follows.

```
3
10
120
3004
```

Thus there are a total of 4 lines in the file, even though the first line specifies three nodes for the measurement data. The global node numbers correspond to the global ID in the exodus input to **Sierra/SD**. The order of the nodes in this list corresponds to the order of the data in the corresponding real or imaginary data files. Other than that, there is no restriction on the ordering of the node numbers (i.e. they do not need to be in ascending or descending order).

Figure 2-1. – Sample Data Truth Table Input for Acoustic Problem

```
4
25  1  1  1
96  1  1  1
13  1  1  1
17  1  1  1
```

Figure 2-2. – Example Data Truth Table for Structures. There are 4 nodes, with numbers 25, 96, 13, 17. The second through fourth columns of 1, 1, 1 mean that all structural degrees of freedom are active, the fifth column is acoustic and implied to be zero.

```
4
5  0  1  0  0
6  0  1  0  0
7  0  0  0  1
8  0  0  0  1
```

Figure 2-3. – Example data truth table for Structural Acoustics. There are 4 nodes, with numbers 5 to 8. Nodes 5 and 6 are structural where only the y component of displacement is measured. Nodes 7 and 8 are acoustic, where only the pressure is measured.

Real Data File

For inverse problems in the frequency domain, the `real_data_file` contains the real component of the measurement data at each frequency, corresponding to the nodes that are specified in the `data_truth_table` file. For a multi-experiment inverse problem, several files must be specified.

An example of the `real_data_file` for an acoustics-only problem is shown in Figure 2-4. The first line of the file contains the number of nodes where measurement data is provided, followed by the number of frequencies of data. Starting on the second line, the real part of the data at the first node is given for all frequencies. In particularly, starting on the second line the data corresponds to the first node in the truth table list, not the node with the lowest ID number. Similarly, subsequent lines contain the real part of the data, at all frequencies, for the remaining nodes. Note that, since this an acoustics-only example, only one line of data is needed for each node in the truth table. The format for the data files describing voltages exactly matches the format of the acoustics-only data file.

An example of the `real_data_file` for a structural-only problem is shown in Figure 2-5. Starting on the second line, the real part of the data at the first node is given for all frequencies. In particularly the first node is the node first in the truth table list, not the node with the lowest ID number. The x displacements for all nodes in the truth table is listed first, followed by the y displacements for all nodes, followed by the z displacements for all nodes.

For viscoelastic material identification problems in the frequency domain with `data_type` set to `moduli`, the `real_data_file` contains the real part of the shear and bulk modulus. An example of the `real_data_file` is shown in Fig. 2-6. The first line of the file is always 2, followed by the number of frequencies of data. Currently, this feature only supports one frequency of data, hence the first row should read 2 1. The second and third lines are respectively the shear modulus and bulk modulus. There is only one column, corresponding to a single frequency of measured data.

The frequencies of the measured data are specified in the **frequency** section. The frequencies given by **frequency** section must correspond to the frequencies where the experimental data was measured. These frequencies can be either uniformly or non-uniformly spaced, as specified in the **frequency** section.

We build on the small example given in Figure 2-1 that has measurements at nodes 10, 120, and 3004, and consider the case where there are 2 frequencies in the data set. The `real_data_file` file for an acoustics-only problem could look as follows

3	2
1.1	2.4
0.7	3.3
2.1	1.4

The actual values in the above table were chosen arbitrarily, but observe that there is one "header row" followed by 3 data rows. There are 2 columns, corresponding to the two frequencies of the measured data. The units of the measurements must correspond to appropriate units in the analysis. One row of data is required for each DOF in the truth table. Data is required even if the value in the truth table is zero. If the truth table value for a DOF is zero, the data is not used in the analysis.

Figure 2-4. – Sample Real Data File Input for an acoustics-only problem

We build on the small example given in Figure 2-2 that has measurements at nodes 25, 96, 13, and 17, and consider the case where there is only 1 frequency in the data set. The `real_data_file` file for a structural-only problem could look as follows

```
12          1
1.1 // x-displacement for node 25
0.7 // x-displacement for node 96
2.1 // x-displacement for node 13
1.1 // x-displacement for node 17
0.7 // y-displacement for node 25
2.1 // y-displacement for node 96
1.1 // y-displacement for node 13
0.7 // y-displacement for node 17
2.1 // z-displacement for node 25
1.1 // z-displacement for node 96
0.7 // z-displacement for node 13
2.1 // z-displacement for node 17
```

The actual values in the above table were chosen arbitrarily, but observe that there is one "header row" followed by 12 data rows. There is only one column, corresponding to the single frequency of the measured data. The units of the measurements must correspond to appropriate units in the analysis. For structures these are units of displacement. Rows $2-5$ correspond to the x components of the displacements at the nodes from the truth table, rows $6-9$ correspond to the y components of displacement, and rows $10-13$ correspond to the z components. Note that the x components of displacements for all nodes are listed first, followed by the y components for all nodes, followed by the z components for all nodes. One row of data is required for each DOF in the truth table. Data is required even if the value in the truth table is zero. If the truth table value for a DOF is zero, the data is not used in the analysis.

Figure 2-5. – Sample Real Data File Input for a structural-only problem

The `real_data_file` file for a viscoelastic material identification problem with `data_type = moduli` could look as follows

```
2          1
5.7 // Shear modulus
3.1 // Bulk modulus
```

The actual values in the above table were chosen arbitrarily, but observe that there is one "header row" followed by 2 data rows.

Figure 2-6. – Sample Real Data File Input for a data type moduli problem

Imaginary Data File

The `imaginary_data_file` has the exact same format as the `real_data_file` except that it contains the imaginary part of the data rather than the real part.

Random PSD Data File

In the context of PSD inversion, instead of specifying the real and imaginary data files, a single `psd_data_file` is needed, which contains the complex (Hermitian) PSD matrices for all the frequencies. The first two numbers in the `psd_data_file` must be the number of frequencies and the number of (measured) response degrees of freedom. The remainder of the file contains the PSD matrices, one matrix each for each frequency. The format of the PSD matrix should be in the natural row-oriented format, with each line containing a row of the matrix. The complex values should be entered in *(real part, imaginary part)* format, with space between the numbers. Sierra-SD automatically checks if each of the PSD matrices is Hermitian and positive definite, as each PSD matrix should be.

Data File

Transient time history data is stored in the `data_file`. The format is identical to the `real_data_file` except for the addition of the time step as the third column in the first row. Each column of data now corresponds to a time step of analysis. An example is shown in Fig. 2-7. No interpolation is performed, and the measured data must exactly match the time steps of the analysis. As in the `real_data_file`, the rows of data are grouped as all of the x components of displacements at the measured nodes, followed by all of the y displacements, followed by the z displacements.

We note that for time-domain inverse problems, the data in the `data_file` must be padded with zeros at the beginning, since the sensor data is typically started at time $t = 0$. The maximum time-of-flight from the input loads to the sensors can be estimated easily (an upper bound is fine). Then, dividing the maximum time-of-flight by the time step gives the number of zeros to be added to the beginning of each time history. Without these zeros, the forward problem would not be able to come up with loads that match the sensor data in the early time response, due to the finite wave propagation speed.

```
6    4    0.1
0.0  0.0  2.1  2.3
0.0  0.0  2.3  3.5
0.0  0.0  3.6  4.1
0.0  0.0  1.5  1.8
0.0  0.0  0.9  1.4
0.0  0.0  3.4  9.5
```

Figure 2-7. – Sample Transient Data File Input for a structural-only problem

Modal Data File

The `modal_data_file` contains measured modal results for inversion with the `eigen` solution. The first line of the file is the number of eigenvalues. Each subsequent line contains only the eigen frequency of the measured mode. It is important that the simulation modes are in the same order as the test modes.

Modal Weight Table

Not all computed eigenvalues may correspond to a test (or measured) mode. Further, even those modes identified may be more or less important. The `modal_weight_table` contains the weights applied to each computed mode.

The first line contains the number of weights, which must match the number of modes computed in the analysis, and each subsequent line contains the weight for the corresponding computed mode. All weights must be non-negative, and computed modes with no corresponding measured data should have zero weight.

Data Weight Table

In the case of eigenvector inversion (described in Sec. 2.3.2), the `data_weight_table` contains weights applied to each computed eigenvector error. This parameter allows for independent weighting of the eigenvalues and eigenvectors, but is not required; by default, it is the same as the `modal_weight_table`.

As above, the first line contains the number of weights, which must match the number of modes computed in the analysis, and each subsequent line contains the weight for the corresponding computed mode. All weights must be non-negative, and computed modes with no corresponding measured data should have zero weight.

2.7.5. Block section for Material Identification

inverse_material_type

For material inverse problems, the **block** section provides an additional option to control the optimization strategy. In particular, blocks can be specified as `known`, `homogeneous`, or `heterogeneous`, as shown in Table 2-11, and an example is shown in input 2.7.4.

The default behavior for the **inverse_material_type** keyword is `known`, which implies that the material parameters given in the corresponding material are fixed, and are not modified in the optimization process. In the case where all blocks are known, there is no need to solve the material inverse problem at all.

The remaining two options for the **inverse_material_type** keyword are:

- `homogeneous`. In this case, the material parameters are unknown and will be optimized during the inversion process, but are treated as constant over the entire block. This option is best used in the case when material properties are unknown in a block, but not expected to vary much over the block.
- `heterogeneous`. In this case, the material parameters are also unknown, will be optimized in the inversion process, but each element in the block will be given its own material properties. This is typically referred to as *spatially varying material properties*.

For a block to be used in the inverse problem, the **inverse_material_type** must be included and specified to either `homogeneous` or `heterogeneous`.

In input 2.7.4 Block 1 is `heterogeneous`, Block 2 is `homogeneous`, and Block 3 is `known`.

Parameter	type	Description
inverse_material_type	string	block inverse type

Table 2-11. – Block Section Parameters for Material Inversion

```
block 1
  material 1
  inverse_material_type heterogeneous
END
block 2
  material 2
  inverse_material_type homogeneous
END
```

```
block 3
  material 3
  inverse_material_type known
END
```

Input 2.7.4. Basic Block Section Example for Material Inversion

joint2g and blkbeta

Additional material parameters that can be optimized include `joint2g` elements and block proportional stiffness damping `blkbeta`. Since these parameters live in the **block** section rather than in the **material** section, their bound constraints are specified in the **block** section. *These blocks must also be marked for inversion by setting* **inverse_material_type** *to be* `HOMOGENEOUS`. If the **inverse_material_type** is not set, the joint2G block will be silently ignored during the inverse routines. An example of input syntax for `joint2g` and `blkbeta` inputs for material optimization are given in Table 2-12 and input 2.7.5.

The `joint2g` elements involve 6 parameters that can be optimized. The `joint_truth_table` specifies which of these parameters will be included in the optimization solution. For example, in input 2.7.5, only the first (i.e. *x-component*) of the `joint2g` parameters will be optimized. The remaining parameters will stay fixed in the optimization, including those that are specified as *NULL*. In the case of `blkbeta`, the optimal *blkbeta* for the specified block will be computed and written to the result file.

Parameter	type	Description
joint_elastic_bounds	real real	bounds on joint2g spring stiffnesses
joint_damper_bounds	real real	bounds on joint2g dashpot (damper) parameters
blkbeta_bounds	real real	bounds on block stiffness proportional damping

Table 2-12. – Block Section Parameters for Material Inversion

```
block 50
  joint2g
  kx = elastic 2.474e5
  ky = elastic 2.e8
  kz = elastic 2.e8
  krx = NULL
  kry = NULL
  krz = NULL
  joint_elastic_bounds = 2.0e5  1e9
  joint_truth_table = yes no no no no no
  inverse_material_type = homogeneous
end
```

```
block 51
  inverse_material_type = homogeneous
  material 1
  blkbeta 2.0e-4
end
```

Input 2.7.5. Block Section Example for `joint2g` and `blkbeta` Material Inversion

2.7.6. Material section for Material Identification

For material inverse problems, the **material** section provides additional options to control the optimization strategy. Currently, for 3D elements, only `isotropic`, `orthotropic`, `isotropic_viscoelastic_complex`, and `acoustic` material types are supported for material inverse problems.

Table 2-13 shows which material parameters can be optimized in the different solution procedures. Details of these parameters are discussed in the following sections and are summarized in Table 2-14. In general, parameters associated with damping can only be optimized in **directfrf** solutions, and those associated with stiffness can be optimized in either **directfrf** of **eigen** .

Parameter	directfrf-inverse	eigen-inverse
K	yes	yes
G	yes	yes
E	yes	yes
Nu	yes	yes
K_real	yes	no
K_imag	yes	no
G_real	yes	no
G_real	yes	no
Cij	yes	no
c0	yes	no
density	yes	no
joint2g elastic	yes	yes
joint2g damper	yes	no
blkbeta	yes	no
spotweld	yes	yes

Table 2-13. – Table of Supported Material Parameters for Inverse Methods

For isotropic, orthotropic, and acoustic materials, the set of unknown material parameters may be customized by explicitly specifying the parameter `num_material_parameters` and a list of strings following `material_parameters`. These strings may include `bulk` and/or `shear` for elastic materials, `orthotropic` for orthotropic elastic materials, `sound_speed` for acoustic

materials, and `rho` for all of the above. Examples of specifying the `num_material_parameters` and `material_parameters` will be shown in the following sections.

Isotropic Material Inversion

For isotropic elastic materials, the user can invert for density and/or any two of K, G, E, or nu. Isotropic materials are the default material type for 3D elements and the keyword **isotropic** is not required. Bounds on the parameters are specified as shown in Table 2-14 and discussed on page 52. Example **material** sections for isotropic inversion are shown in input 2.7.6. Note that the parameters `G_bounds`, `K_bounds`, `E_bounds`, and `Nu_bounds` only apply to isotropic materials. Additionally, note that K and G are the default material inversion parameters. To invert for E and/or nu, specify the `num_material_parameters` and `material_parameters`, as seen in input 2.7.6.

```
material 1
  isotropic
  G_bounds 0 1e4
  K_bounds 0 1e4
  G 100.0
  K 200.0
  density 10.0
end
material 2
  isotropic
  E_bounds 1e-10 1e40
  Nu_bounds -0.9999 0.49999
  E 1.0e6
  Nu 0.25
  density 10.0
  num_material_parameters = 2
  material_parameters = youngs poissons
end
```

Input 2.7.6. Material Section Example for Isotropic Material Inversion

Orthotropic Material Inversion

Unlike isotropic inversion, orthotropic inversion requires special care with respect to inadvertent material instabilities, i.e., during the inversion iterations, the elasticity tensor may not satisfy positive definiteness, potentially leading to the failure of even the forward solution. To avoid this, we parametrize the material tensor using the standard normal moduli E_{ii}, shear moduli G_{ij}, and special dimensionless parameters $A_{ij} = E_{ij}/\sqrt{E_{ii}E_{jj}}$. This is referred to as *Alpha* parametrization. Such parametrization lends to easier imposition of material stability constraint

which is done through a single inequality constraint and several bound constraints. Further details will be provided in an upcoming SAND report. Notwithstanding the theoretical details, the user is asked to pay special attention to the definition of A_{ij}, when applying bound constraints on these parameters. Bound constraints can be specified as as shown in Table 2-14, and an example is shown in input 2.7.7. Note that the parameters `Eij_bounds`, `Gij_bounds` and `Aij_bounds` apply only to orthotropic elastic materials. These parameters constrain the optimizer to work in a restricted space of possible design variable values, and prevent convergence to physically unrealistic values of the parameters. An alternative to Alpha parametrization is *Cholesky* parametrization, where the modulus matrix is parametrized through Cholesky factorization, avoiding the need of inequality constraints. This can be utilized by setting the flag `alphaparametrization` to `no`.

Transverse Isotropic Material Inversion is performed essentially through orthotropic inversion, by setting the flag `transverselyisotropic` to `yes`, followed by the plane of isotropy represented by a two-digit number, i.e., 12, 23, or 13, where 1,2,3 represent the three axes of material symmetry consistent with the coordinate system within the element block. Input 2.7.7 contains an example for transversely isotropic inversion.

```
material 3  // Full orthotropy
  density = 1.0
  orthotropic
  Cij = 4.0 1.0 2.0
  5.0 3.0
  6.0
  2.0
  1.0
  3.0

  Eii_bounds = 0.01 20.0 0.01 20.0 0.01 20.0
  Gij_bounds = 0.01 10.0 0.01 10.0 0.01 10.0
  Aij_bounds = 0.0 0.707 0.0 0.707 0.0 0.707
end

material 4 //Transeverse isotropy
  density = 1.0
  orthotropic
  alphaparametrization  no
  inequalityconstraints no
  transverselyisotropic yes 23
  Cij = 5  1  1
    16  6
  16
  5
  1
  1
```

```
  Eii_bounds = 1 100 1 10 1 100
  Gij_bounds = 0.1 5 0.1 5 0.1 5
end
```

Input 2.7.7. Material Section Example for Orthotropic Material Inversion

Viscoelastic Material Inversion

For isotropic viscoelastic materials, the user inverts for the real and imaginary components of the shear and bulk moduli, G_{real}, G_{imag}, K_{real}, and K_{imag}. Because these four components are frequency-dependent, the initial guesses are specified in functions that define the values of these parameters as a function of frequency. As shown in input 2.7.8, in this example functions $1-4$ specify the initial guesses for the real and imaginary components of G and K. Note that `Greal_bounds`, `Kreal_bounds`, `Gim_bounds`, and `Kim_bounds` only apply for viscoelastic materials.

The `directfrf-inverse` method supports viscoelastic material identification using homogenized moduli data by setting `data_type` to `moduli` in the **inverse-problem** block (see section 2.7.4). This feature is only available for the `inverse_material_type homogenous` option. In addition, all material blocks, including known materials, must be specified as `isotropic_viscoelastic_complex`.

```
material 5
  isotropic_viscoelastic_complex
    Greal_bounds 0 1e4
    Kreal_bounds 0 1e4
    Gim_bounds 0 1e2
    Kim_bounds 0 1e2
  Greal = function 1
  Gim = function 2
  Kreal = function 3
  Kim = function 4
  density = 10.0
end
```

Input 2.7.8. Material Section Example for Viscoelastic Material Inversion

Acoustic Material Inversion

For acoustic materials, the user may invert for sound speed c_0 and/or density. Alternatively, the parameter `impedance_match` may be specified. This has the effect of optimizing for sound speed c_0 and density ρ_0 under the condition where impedance $Z = \rho_0 c_0$ is constant. Note that the parameters `impedance_match` and `c0_bounds` apply only to acoustic materials.

Material Bounds

Bounds on the material parameters are specified using a keyword for each parameter. Generally, this keyword will have the form <*parameter*>_bound. The lower and upper bounds are then specified as the first and second numbers, respectively, to follow this keyword. For example, in input 2.7.6, E is restricted to be within 1e-10 and 1e40. A full list of the keywords that can be specified in the **material** section, including those used to define bounds, is given in Table 2-14.

Important: The optimizer considers the endpoints of the bounds to be within the set of valid parameter values. Therefore, the specified bounds must also be physical. As an example, if the user specifies `Nu_bounds = 0.0 0.5`, then it is possible that *nu* will be evaluated at 0.5 and **Sierra/SD** will throw a fatal error. As an alternative, the user may specify `Nu_bounds = 0.0 0.49999` and avoid early termination of the optimization due to this error. Similarly, E, G, and K must be strictly positive. Instead of a lower bound of 0, the user should set a lower bound of 1e-10 or some similarly small number.

Note that `boundConstraints = yes` is required for *all* material-ID type problems and will be activated automatically for `design_variable = material`, `design_variable = multi_material`, `design_variable = damage`, etc., regardless of the user-defined value for `boundConstraints`.

Parameter	type	Description
G_bounds	real real	lower and upper bounds on shear modulus
K_bounds	real real	lower and upper bounds on bulk modulus
E_bounds	real real	lower and upper bounds on Young's modulus
Nu_bounds	real real	lower and upper bounds on Poisson's ratio
Greal_bounds	real real	lower and upper bounds on real part of shear modulus
Kreal_bounds	real real	lower and upper bounds on real part of bulk modulus
Gimag_bounds	real real	lower and upper bounds on imag part of shear modulus
Kimag_bounds	real real	lower and upper bounds on imag part of bulk modulus
Eij_bounds	6 reals	lower and upper bounds on the three normal moduli
Gij_bounds	6 reals	lower and upper bounds on the three shear moduli
Aij_bounds	6 reals	lower and upper bounds on the three α parameters
density_bounds	real real	lower and upper bounds on density
c0_bounds	real real	lower and upper bounds on sound speed
impedance_match	real	value of $\rho_0 c_0$ to match for acoustic materials
num_material_parameters	int	(optional) number of material parameters
material_parameters	list of strings	(optional) `bulk`, `shear`, `rho`, `orthotropic`, and `sound_speed`

Table 2-14. – Material Section Parameters for Material Inversion

Material Initial Guess

Initial guesses for the material parameters are also given in the **material** block. In general, if the **block** section does not define the `inverse_material_type` as `known`, then the initial guess will be taken from whatever parameter values are given in the **material** block. As a concrete example, in the case of isotropic elastic materials the initial guess is taken as the values for G and K (or alternatively, E and nu) that are included in the **material** block. In the example shown in input 2.7.6, the shear and bulk moduli are given initial guesses of 100 and 200, respectively.

Damage Identification

Damage identification is a design variable implemented for `directfrf-inverse` problems as a special case of elastic material identification. In damage identification, a damage phase field variable is used to interpolate between damaged (weak) and full-strength material, ideally converging to near-binary values to indicate presence of material damage. Damage ID is activated by `design variable = damage` in the **inverse-problem** block.

Damage identification employs techniques originated for topology optimization problems but may used for a variety of applications, including identification of weakened material regions, determination of contact area in a thin layer of elements at an interface, or two-phase material design. The elastic damage model uses a Solid Isotropic Material with Penalization (SIMP) model [3] that interpolates the isotropic elastic and mass density properties in the unknown material between two phases using the scalar phase field variable $\beta \in [0, 1]$, expressed as

$$G(\beta) = G_0 + (G_u - G_l)\beta^p \tag{2.7.1}$$

$$K(\beta) = K_0 + (K_u - K_l)\beta^p \tag{2.7.2}$$

$$\rho(\beta) = \rho_0 + (\rho_u - \rho_l)\beta^q, \tag{2.7.3}$$

where $\{G_u, K_u, \rho_u\}$ and $\{G_l, K_l, \rho_l\}$ are the upper and lower bounds for the bulk modulus, shear modulus, and density, respectively. Definition of different powers $p \geq q \geq 1$ for the elastic and mass density components renders elastic properties relatively weaker, for a given mass density, thus disincentivizing intermediate density values. Powers p and q are specified with `penalizationElasticity` and `penalizationMass`, respectively.

Bounds of the material interpolation are specified by setting limits for shear modulus `G_bounds`, bulk modulus `K_bounds`, and mass density `density_bounds` in the **material** block. Meanwhile, the initial damage phase field value is controlled using the `G`, with respect to `G_bounds`, as

$$\beta^{(0)} = \frac{(G - G_l)}{(G_u - G_l)}. \tag{2.7.4}$$

Initial `density` and `K` values must still be given, though are not used to determine the initial phase field value. The solution for β is not available for output, but rather the material parameters are output as evaluated within the penalized elastic and mass density models using the phase field solution.

Damage identification is enabled for both homogeneous and hetereogeneous unknown material blocks. In the homogeneous unknown material block case, only the penalization powers for the mass density and elastic properties must be specified. In heterogeneous unknown material blocks, additional filtering and projection operations are employed to control solution length scale and to encourage binary quality. Kernel filtering prevents development of mesh dependent or checkerboard (i.e. alternating 0-1 phase field density) patterns. In this strategy, a weighted kernel is convolved with the density field, producing a locally-averaged filtered field. The filter kernel radius is defined in the **inverse-problem** block as `filter_radius` (default = 0.1), which should be set no smaller than the minimum distance between element centroids in the unknown material block.

A Heaviside-approximating function is then used to map filtered values closer to 0 or 1 bounds and recover a more-binary valued field. The smooth Heavisde function is defined

$$\tilde{\beta}(\beta) := \frac{\tanh(\zeta\eta) + \tanh(\zeta(\beta - \eta))}{\tanh(\zeta\eta) + \tanh(\zeta(1 - \eta))}. \tag{2.7.5}$$

Here, the slope of the smooth Heaviside $\zeta > 1$ is specified by `smooth_heaviside_slope`, while its inflection threshold $\eta \in (0, 1)$ is specified by `smooth_heaviside_threshold`. Typically, modest values for `smooth_heaviside_slope` (5-10) can produce high contrast fields without creating an excessively severe projection, which can impede optimization convergence.

Below, we summarize the necessary parameters in the **material** and **inverse-problem** sections for damage identification problems.

Parameter	type	Description
G	real	initial value of shear modulus (determines initial phase field value)
K	real	initial value of bulk modulus
density	real	initial value for mass density
G_bounds	real real	lower and upper bounds on shear modulus interpolation
K_bounds	real real	lower and upper bounds on bulk modulus interpolation
density_bounds	real real	lower and upper bounds on mass density interpolation

Table 2-15. – Parameters in **material** section for Damage Identification

Parameter	type	Description
design_variable	string	damage
penalizationElasticity	integer	elasticity penalty exponent (default = 3)
penalizationMass	integer	elasticity penalty exponent (default = 1)
filter_radius	real	filter kernel radius (default = 0.1)
smooth_heaviside_slope	real	smooth heaviside projection slope (default = 1)
smooth_heaviside_threshold	real	smooth heaviside inflection threshold (default = 0.5)

Table 2-16. – Parameters in **inverse-problem** section for Damage Identification

Spot Weld Stiffness Inversion

Material parameter identification techniques can be extended to calibrate contact stiffness by employing the spot weld construct in **Sierra/SD**. Spot welds are virtually-constructed element blocks at a contact interface, in which node-face interactions are assigned stiffnesses in normal and tangential directions. Spot weld elements offer multiple benefits for representing contact: they are minimally instrusive from the model construction perspective, they can provide spatially-varying, tunable interface stiffness, and they are more scalable than tied data constraints, as they are stored on the subdomain level. In SD inverse problems, these normal and tangential stiffness parameters can be updated to improve the model's match to supplied target or experimental data.

Spot weld stiffnesses can be enabled as design variables by specification of several keywords in the spot weld block definition. In the material identification solution context, the normal and tangential stiffness parameters are treated as independent design variables. Syntax for spot weld stiffness inversion involves including a few keywords in the conventional spot weld block definition:

- Set **inverse_material_type** as either **heterogeneous** , **homogeneous** , or **known** . **Heterogeneous** spot welds treat stiffnesses of each spot weld element as independent design variables, whereas **homogeneous** spot welds have uniform stiffness parameters assigned to all elements. **Known** treats the spot weld stiffnesses as known and deactivates the design variables.
- Set initial stiffness guesses with **normal displacement scale factor** and **tangential displacement scale factor** keywords.
- Define bounds on the normal and tangential stiffness values using keywords **normal displacement scale bounds** and **tangential displacement scale bounds**. Lower bound values must be positive and default to zero if not supplied.

Refer to Input 2.7.9 for an example definition of spot weld design variables. Currently, spot weld inversion is tested for directfrf-inverse and eigen-inverse solution cases. For additional spot weld syntax suggestions, refer to the **Sierra/SD** User's Manual [21]. Note, as with conventional spot welds, it is recommended to define a linear function with positive slope for the **normal displacement function** and **tangential displacement function** keywords and allow the optimizer to adjust the scale factors further.

Include keyword **spot_weld** in the **OUTPUTS** section to export spot weld stiffness solutions, which are written to the **spot_weld_norm_stiffness** and **spot_weld_tang_stiffness** variables in the output exodus file. Additionally, for **homogeneous** design variables, values are reported to the DesignVariableIterations.txt file at each iteration. If desired, the user can also enable the output of heterogeneous design varibles to exodus at each iteration. See section 2.7.6.1 for details.

```
INVERSE-PROBLEM
  modal_weight_table = val_weights.txt // data for eigen-inverse solution
  modal_data_file = val_data.txt
  data_truth_table = vec_ttable.txt
  data_file = vec_data.txt
  eigen_objective = matching
  design_variable = {material | damage}
  penalizationElasticity = 4 // penalty parameter for damage
END


BEGIN SPOT WELD
    sideset = side_1
    second surface = side_2
    search tolerance = 0.01
    normal displacement function = y_equals_x
    normal displacement scale factor = 5e1  // initial guess
    tangential displacement function = y_equals_x
    tangential displacement scale factor = 5e1 //initial guess
    inverse_material_type = {known | homogeneous | heterogeneous}
    normal displacement scale bounds =  1e0 1e3
    tangential displacement scale bounds = 1e0 1e3
END

FUNCTION y_equals_x
  type analytic
  evaluate expression “x”
END
```

Input 2.7.9. Example definition of spot weld for stiffness inversion

Spot Weld Damage Identification

In contact calibration or debond detection problems, identification of regions with full contact or no contact is often desired. These scenarios can be addressed by using a “damage" parameterization of the spot weld stiffness: here, the normal and tangential stiffnesses are connected to a phase-field variable which is encouraged to converge to binary (i.e. 0 or 1) values that represent soft or stiff contact. We employ a Rational Approximation of Material Properties (RAMP) model to evaluate the normal and tangential stiffnesses, using the phase-field variable to interpolate between the lower and upper stiffness bounds. For phase-field parameter $\beta \in [0, 1]$, the stiffness property $k = k(\beta)$ is evaluated

$$k(\beta) = k_l + (k_u - k_l)\frac{\beta}{1 + p(1 - \beta)} \tag{2.7.6}$$

for bound values $\{k_l, k_u\}$ and penalty parameter $p > 0$ (we suggest $p \geq 3$).

This parameterization of the spot weld stiffness encourages convergence to binary stiffness fields for multiple reasons. As the tangential and normal stiffnesses are connected together, an update to one stiffness direction informs the value of the other stiffness direction, preventing disagreement between the normal and tangential stiffness fields. Furthermore, the RAMP model reduces the relative stiffness of intermediate phase-field values, allowing only phase-field values near 1 to attain full stiffness. Finally, the dimensionality of the design variable is reduced under the condensed phase-field parameter, which tends to improve convergence behavior.

Damage parameterization of the spot weld stiffness is specified with similar syntax to the material damage parameter (see Table 2-16):

- Set **design_variable = damage** to select the damage parameterization for spot weld blocks.
- Define the penalty parameter p in Eq. 2.7.6 with **penalizationElasticity=<double>** . The default value is 3.
- Set the initial guess for a spot weld damage block by defining the initial **normal displacement scale factor** ; the initial phase-field value is inferred as the inverse of the RAMP model for the initial normal stiffness. The tangential stiffness value is then evaluated using the RAMP model and the initial phase-field value.

Example syntax is also shown in Input 2.7.9.

2.7.6.1. Inverse-SD Design Variable Output

The file "DesignVariableIterations.txt" summarizes the design variable values at each optimization iteration. Inverse materials that are flagged as homogeneous will be written to this file at each iteration by default. This ensures that design variables are saved in case the optimization run terminates unexpectedly. This process is automatic and requires no additional input from the user.

If any materials are flagged as heterogeneous then all design variables will be written to the Exodus output only if specified in the XML file. This is in addition to the homogeneous design variables that are written to "DesignVariableIterations.txt". Currently, only eigen-inverse material identification cases are supported, which include spot weld inversion, damage identification, and heterogeneous material inversion. The contents of the Exodus file depend on the specifications provided in the OUTPUT, HISTORY, and FREQUENCY SD input deck blocks. All output variables specified in these blocks will be written at each optimization iteration.

To enable heterogeneous outputs to Exodus, include the parameter "Exodus Output Interval" in the XML file. This parameter should be an integer that specifies the interval at which the file will be written. For example, setting this parameter to one will write the Exodus file at every iteration, while setting it to two will write the Exodus file at every other optimization iteration (beginning with ROL iteration 1, not 0). If the user wishes to turn off this capability, set the interval to a large value. This feature is disabled by default.

An example of this option is shown below

```
〈 ParameterList name="Inputs"〉
  〈 Parameter name="Exodus Output Interval" type="int" value="2" 〉
〈/ ParameterList〉
```

Input 2.7.10. Example of Exodus Output Interval

2.7.7. Loads section for Load Identification

For inverse load problems, the **loads** section provides additional options to control the optimization strategy. Inverse loads are currently supported for acoustics and structures. Inverse acoustic loads are only supported for sidesets. Inverse structural loads are supported on both sidesets and nodesets. On sidesets, both pressures and traction loads may be optimized in the inverse problem. For nodesets, both forces and moments can be optimized, the latter only being applicable in the case when the nodeset in question has rotational degrees of freedom (e.g. a concentrated mass). Table 2-17 summarizes the load options which apply to inverse methods. The `inverse_load_type` options are detailed in Table 2-18.

The default behavior for the `inverse_load_type` keyword is `known`, which implies that the loads on that sideset or nodeset are fixed, and are not modified in the optimization process. In the case where all blocks were known, there would be no need to solve the inverse loads problem.

In input 2.7.11 sideset 1 is `known`, sideset 2 is a real-valued, unknown acoustic load that is `spatially_constant`, and sideset 3 is an imaginary-valued, unknown acoustic load that is `spatially_constant`. We note that in the case of a transient problem, the **loads** block in an inverse loads problem would look the same except that there would be no imaginary loads in that case.

For acoustic problems currently inverse acoustic loads are limited to the `acoustic_accel` option. The `acoustic_vel` keyword is not supported for acoustic loads.

Input 2.7.12 shows a similar example for a structural pressure, traction and force load case. This example contains a known pressure load, and unknown pressure, traction, and force loads. Note that in the case of traction and force loads, the direction of the traction load (in this case 111) is fixed, and only the function amplitudes are calculated in the inverse problem.

In input 2.7.12, functions 1 – 4 contain the initial guesses for the load amplitudes for sidesets 1 – 3 and forces/moments on nodeset 4. These load amplitudes are then refined during the optimization process. The resulting loads are written to a text file called *force_function_data.txt*, which could then be included in a subsequent forward or inverse loads case in a restart analysis.

Parameter	**type**	**Description**
inverse_load_type	string	load inverse type

Table 2-17. – Loads Section Parameters for Force Inversion

Parameter	**Description**
spatially_constant	Load amplitude is unknown and will be optimized during the inversion process, but is treated as constant over the entire sideset or nodeset.
spatially_variable	Load amplitude is unknown and will be optimized during the inversion process. Each dof on the sideset or nodeset is optimized.
known	default. No optimization performed.

Table 2-18. – Inverse Load Type Options

```
loads
  sideset 1
    acoustic_accel = 1
    function = 1
  sideset 2
    acoustic_accel = 1
    function = 2
    inverse_load_type = spatially_constant
  sideset 3
    iacoustic_accel = 1
    function = 3
    inverse_load_type = spatially_constant
end
```

Input 2.7.11. Loads Section Example for Acoustic Force Inversion

```
loads
  sideset 1
    pressure = 1
```

```
    function = 1
  sideset 2
    pressure = 1
    function = 2
    inverse_load_type = spatially_constant
  sideset 3
    traction = 1 1 1
    function = 3
    inverse_load_type = spatially_constant
  nodeset 4
    force = 0 1 1
    function = 4
    inverse_load_type = spatially_constant
  nodeset 4
    moment = 0 1 1
    function = 5
    inverse_load_type = spatially_constant
end
```

Input 2.7.12. Loads Section Example for Structural Force Inversion

2.7.8. Limitations for Inverse Load Problems

There are a number of limitations which apply to transient load identification. These include the following.

1. Structural, acoustic and structural-acoustic domains may be addressed.
2. Only the simple Newmark integrator should be used, i.e. do not use generalized alpha integration and do not use the `rho` keyword.
3. Pressure is always applied along the surface normal, and tractions are applied along the direction specified in the **loads** block. Inverse methods do not support follower pressures.
4. Load identification applies to `acoustic_accel` loadings on acoustic sidesets, pressures/tractions on structural sidesets, and forces and moments on structural nodesets.
5. In force identification problems, a pressure, traction or force may be applied on a shell. The measured displacement fields in the truth table for shells can be applied to nodes with rotational DOFs; however, only displacement DOFs can be specified in the data files. Specifying rotational DOFs as measured data at nodes is not supported.

2.7.9. ROL Output for Inverse Problems

Sierra/SD uses the Rapid Optimization Library (ROL), which is part of Trilinos [9] for solving optimization problems. During the optimization process, ROL writes out a text file called *ROL_Messages.txt* that contains information about the convergence of the optimization solution. It is important to examine this file to assure that the solution is adequately converged.

An example of a *ROL_Messages.txt* is given in Figure 2-8. The first 2 lines show the optimization method that was used by ROL, and the following lines contain convergence information. Each line corresponds to a single optimization iteration. The first column shows the iteration number under the *iter* heading. The second column shows the value of objective function at that iteration, under the *value* heading. The third column shows the absolute norm of the gradient (i.e. the derivative of the objective function with respect to the optimization variables). In Figure 2-8 we only show the first three columns of output as these will typically be of most interest to the user, but the remaining columns contain information about step size, number of function evaluations, etc... (Denis, Drew, any input here would be great).

For a typical **Sierra/SD** user, the first three columns in the *ROL_Messages.txt* file will typically be of the most important to pay attention to. As the goal is to minimize the objective function, a substantial decrease in the second column should be observed. Also, the desired minimum of the optimization corresponds to a zero gradient, and thus the third column should be observed to be as close to zero as possible.

```
Newton-Krylov with Cubic Interpolation Linesearch satisfying
Null Curvature Condition Krylov Type  Conjugate Gradients
iter                 value                gnorm
0                    5.171112e-03         7.337197e-03
1                    1.160506e-10         1.245979e-06
2                    1.453148e-17         4.667996e-10
```

Figure 2-8. – Example of *ROL_Messages.txt* file for Inverse Problem Solution

2.8. Example Inverse Problems

Inverse problems are class of problems where some portion of the solution to an analysis is known, but the inputs to the problem are not. Inverse methods solve an optimization problem where the inputs are optimized to match the solution. The current types of input problems supported are Load ID (transient or FRF) and Material ID (Eigen and FRF).

2.8.1. Experimental Data

For inverse problems, experimental data is typically gathered in a lab. In acoustics, microphones are used to measure acoustic pressure For the transient case, these are measured over a period of

time. For the FRF case, these are measured over a series of frequencies. Introducing additional measurements at new data points generally improves the fidelity of the computed solution. On the other hand, the computational difficulty of solving the inverse problem increases too. For this demonstration, synthetic data is generated by solving a forward acoustic problem.

2.8.2. Inverse Problems - Load-ID

2.8.2.1. Experimental Model

The experimental model is shown in Figure 2-9. The *Football Model* is an ellipsoidal acoustic mesh, with a cylindrical hole in the middle. 70 sidesets are placed around the exterior of the football, allowing for different loading on each sideset.

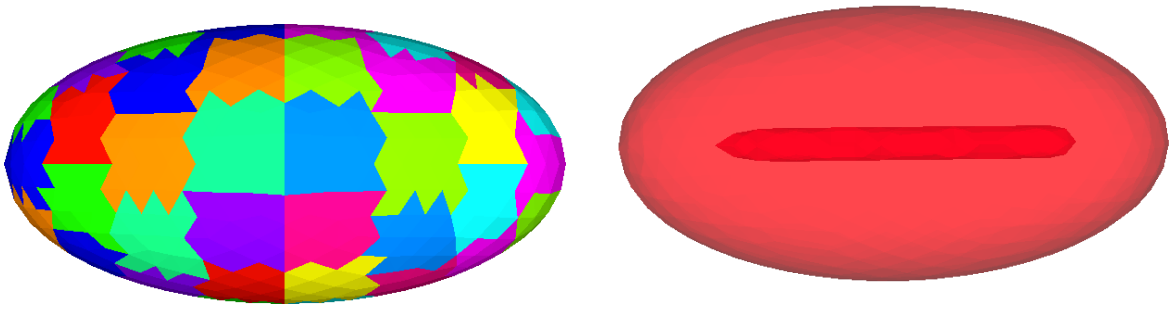


Figure 2-9. – Inverse Football Problem Geometry. On the left, the sideset definitions on the surface. On the right, the interior of the problem.

2.8.2.2. Forward Problem

To generate the experimental data, the model is solved with the solution method `direct-frf`. A set of human generated loads is used to generate “experimental" pressures at a select set of nodes. Any set of loads can be used. The Matlab function `inputDataProcDynFreqAcoustic_Exo.m` is used to generate the experimental data files: `ttable.txt`, `dataReal.txt`, and `dataImag.txt`. This Matlab function requires the results from the forward run, and a nodeset containing the nodes of interest. In practice, this data is generated experimentally, with the measured acoustic pressures being inserted in dataReal.txt and dataImag.txt.

```
solution
        directfrf
    end

    loads
        sideset 4
        acoustic_accel = 1.0
        scale = 2
        function = 10001
```

```
        inverse_load_type = spatially_constant  // ignored
    end
```

Inverse keywords are ignored when running a forward problem.

2.8.2.3. Inverse Problem with known loads

Next, the experimental model is solved with solution method `directfrf-inverse`. and `design_variable = load`. For the first run of the inverse problem, the synthetic loads were left in place, as the “initial guess". The inverse problem converges on the first iteration, as the initial guess is the exact solution to the inverse problem. This is an easy way to make sure the input file are correct. The relative tolerance is shown in the first column of ROL_Messages.txt, and the absolute tolerance is shown in the second column of ROL_Messages.txt. If the exact loading is used as the initial guess, the relative error norm should be on the order of machine precision.

```
solution
    directfrf-inverse
 end

 optimization
   xml_file = rolInput.xml
 end

 inverse-problem
    design_variable = load
    data_truth_table = ttable.txt
    real_data_file = dataReal.txt
    imaginary_data_file = dataImag.txt
 end
```

2.8.2.4. Inverse Problem with unknown loads

Next, the synthetic loads are removed, and the initial guess for the loading is set to be 0 at all time steps. The inverse problem converged in four iterations, with an objective tolerance of 10^{-4}. The objective norm is a relative measure, and any objective norm of 10^{-6} or smaller is considered more than sufficient. Alternatively **opt_tolerance** can be used to set the absolute tolerance. Recommended values for **opt_tolerance** are problem dependent.

2.8.2.5. Verification

Finally, the loading output from the inverse run, `force_function_data.txt`, is used to run the forward problem again. This file is designed so that it can replace the function file with no changes. The problem is verified by checking the pressures at the selected nodes against the initial

run. Though the loading may not be exactly the same between the initial forward run and the verification forward run, the inverse problem has been solved successfully, as the objective function has been solved to the selected tolerance. To generate loading closer to the initial loading, more nodal data can be added or tolerances can be tightened.

2.8.3. Inverse Problems - Material-ID

2.8.3.1. Experimental Model

The experimental model is a solid assembly of two steel blocks joined by a region of viscoelastic foam material. Figure 2-10 shows the geometry of the test model.

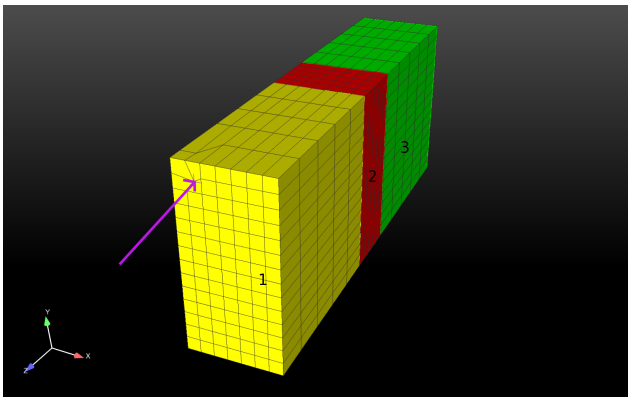


Figure 2-10. – Foam block model with finite element mesh and force location

As shown in Figure 2-10, the model assembly consists of two equally-sized steel blocks, depicted in yellow and green, joined by a region of viscoelastic foam material, shown in red. The model was discretized with a finite element mesh of Hex-8 elements. A periodic point load with a frequency of 500 Hz was applied to the yellow block, also as shown in the figure. It was desired to calculate the frequency-dependent viscoelastic material properties of the foam block, including complex values for the bulk (K) and shear (G) moduli.

2.8.3.2. Inverse Problem input format

The relevant sections of the input used for this example are shown below, followed by some notes about each section.

```
solution
  directfrf-inverse
end
inverse-problem
  design_variable = material
  data_truth_table = ttable.txt
  real_data_file =  data.txt
```

```
  imaginary_data_file =  data_im.txt
end
optimization
  xml_file = rolInput.xml
end

...

block 1
  inverse_material_type = homogeneous
  material 4
  hex8f
end

...

material 4
  isotropic_viscoelastic_complex
  Greal_bounds  -1000 100000
  Kreal_bounds  -1000 100000
  Gim_bounds -1000000 1000000
  Kim_bounds -1000000 1000000

  Greal = function 2
  Gim   = function 3
  Kreal = function 4
  Kim   = function 5
  density=0.010804
end
```

- **solution section:** defines the type of solution (inverse DirectFRF).
- **inverse-problem section:** specifies the design variable (material) and connects externally defined data describing the test.
  - The **data truth table file** contains the global node numbers where the experimental data measurements are given. For example, the input below gives the number of nodal locations (1) in the first line, followed by the single node id (212) showing all structural dof active (1 1 1), and an inactive acoustic dof (assumed 5th column = 0).

    ```
    1
    212    1 1 1
    ```

  - The `real_data_file` and `imaginary_data_file` contain the real and imaginary parts of the measurement data at each frequency. For example, the input below (from a

real data file) gives the number of nodes (3) and frequencies (2), followed by the data at each node. Each frequency requires a separate column of data.

```
3  2
-4.385640897908e-02     -3.985611576838e-02
2.898761003889e-02     3.478175302036e-03
-1.167004279970e-01     -8.319040683879e-02
```

- **optimization section:** provides options to control the optimization strategy. See the users manual for more information on available optimization options.
- **block section:** provides an information on the material of the block. Options include,

  known - The material parameters of the block will not be varied in the inverse solution.

  homogeneous - Material properties are uniform within the block, and are varied to arrive at the best fit for the data.

  heterogeneous - Material properties vary element by element within the block, and are varied to arrive at the best fit.
- **material section:** provides additional options to control the optimization strategy.

2.8.3.3. Running the Inverse Problem

Next, the experimental model is solved as indicated above. A good choice for the first run of the inverse problem is to use the actual material data used as the “initial guess". This causes our problem to converge much faster than with a general guess, and is a good verification step for problems where the material data is known *a priori*. Next, initial guesses for the material data is set to be something other than the actual value to represent a typical initial guess. Convergence data can be found in the file `ROL_Messages.txt`. The objective norm is a relative measure, and any objective norm of 10^{-6} or smaller is sufficient. Alternatively **opt_tolerance** can be used to set the absolute tolerance.

2.8.3.4. Verification

For the problem presented here, the following material data is obtained from running the inverse problem (taken from the `name_0.rslt` file):

```
...

Block 1 Viscoelastic Material Properties
  Real Part of K:    40000.002012
  Imaginary Part of K:    -0.005544
  Real Part of G:    15999.999313
  Imaginary Part of G:    5000.000812
```

```
...
```

This gives us values that we can then use as part of an input for a forward problem, and see if we obtain the same values given in the input data from the truth table and data files. Though the displacements may not be exactly the same between the initial forward run used to generate the inverse data files and the verification forward run, the inverse problem has been solved successfully, as the objective function has been solved to the selected tolerance. To generate more exact results, more nodal data can be added or tolerances can be tightened.

2.8.3.5. Design Variables History Output

A file called `DesginVariableIterations.txt` is generated by each material identification problem. The file contains changes in the values of each design variable throughout ROL iterations. The file has several purposes, in addition to giving user an opurtunity to monitor the progress of inversion algorithms, user can also restart inversion from any given ROL iterations. An example of the output file for viscoeleastic material model inversion is given below.

```
ROL iteration: 0
Block 1 Viscoelastic Material Properties
   Real Part of K:    30000.000000
   Imaginary Part of K:    0.000000
   Real Part of G:    13000.000000
   Imaginary Part of G:    0.000000


ROL iteration: 1
Block 1 Viscoelastic Material Properties
   Real Part of K:    29999.999988
   Imaginary Part of K:    0.000020
   Real Part of G:    12999.999932
   Imaginary Part of G:    0.000119

...
ROL iteration: 81
Block 1 Viscoelastic Material Properties
   Real Part of K:    40000.002797
   Imaginary Part of K:    0.000000
   Real Part of G:    15999.999549
   Imaginary Part of G:    4999.999960
```

This page intentionally left blank.

3. INVERSE METHODS WITH INVERSEARIA

3.1. Introduction

Inverse Aria enables solving inverse problems with SIERRA/Aria using adjoint-based gradients through an interface to the Rapid Optimization Library (ROL). The major advantage of calculating gradients with adjoints comes in the form of computational savings as the design space grows in size. Only one forward solve and one adjoint solve are required to compute the gradient of the reduced objective function with respect to the design variable vector regardless of the number of design variables. Contrast this with finite difference gradients where N+1 forward solves are required for N design variables.

3.2. Outline

- Overview of inverse heat transfer capabilities
- How to build and run
- Inverse problems with example inputs
  - Thermal conductivity
  - Boundary heat flux
  - Contact resistance
  - Arrhenius source term (with FD gradients)

3.2.1. Beta Capabilities and Limitations

Inverse Aria is a still in early development and should be treated as a beta feature. There are three over-arching problem types that Inverse Aria targets with each in various states of development. Currently, only problems with the energy equation are supported by the adjoint solver. The three classes of target problems are material property, boundary condition, and heat source inversion. For the first two years of development (FY20 and FY21), Inverse Aria has been limited to solving linear conduction problems. Typically, thermal engineering problems of interest contain non-linear effects such as temperature dependent material properties, chemical decomposition, and thermal radiation among others. In FY22, development shifted from adding new inverse design variables to focusing on supporting non-linear heat conduction physics.

Thermal conductivity was the first target material property to be developed. Currently, thermal conductivity inversion is possible in Inverse Aria for basic (non-porous) materials with a limited number of boundary conditions. The user can invert for conductivity on a block by block or element by element basis (see Sec. 3.4).

Heat flux to a surface was the first target boundary condition inversion problem. Both steady and transient heat flux inversion are available in Inverse Aria as described in Sections 3.5 and 3.6. Another important boundary condition in thermal problems occurs at the interface between two materials. Typically, this represented by a Robin boundary conditions characterized by a thermal contact resistance or its inverse, thermal gap conductance (Sec. 3.7).

Specific combinations of parameters can be inverted for at once. Heat flux and contact resistance on different boundaries can be solved together. To solve for a combination of thermal conductivities and fluxes or contact resistances, the unknown conductivity block must not touch any of the unknown surfaces.

Many real problems involve at least one reacting material. Examples include pyrolysis of organic materials, ablation of thermal protection systems, and thermal runaway of batteries. These reactions can add or remove heat from the system and are represented in Aria as volumetric heat source/sink terms. Typically, these reactions can be model by Arrhenius forms. To achieve wider ranging applicability, Inverse Aria must be able to solve problems with reacting source terms, either inverting directly for the source term model parameters or inverting for other parameters (material or boundary conditions) in problems where reacting materials are present. As a first step towards enabling this functionality, inversion for the Arrhenius activation energy and frequency factor with finite-difference gradients has been enabled in Inverse Aria. Development of this feature serves to put the building blocks in place for solving this class for problems while further research is conducted on deriving and implementing the adjoint solve.

Computation of the objective function requires experimental data and will typically benefit from some form of regularization. Inclusion of experimental data is currently limited to time histories of temperatures at user specified node locations.

A finial limitation is that problem size is currently constrained by available memory. To execute an adjoint solve, all state variables at every node and time step must be saved in memory. This constraint will be alleviated in future releases by using checkpointing schemes such as Wang et al. [24].

3.2.2. Getting Started with Inverse Aria

Inverse Aria is included in the SIERRA module as of 5.17. It can be loaded with

```
$  module load sierra
```

and run as follows

```
$  inverse_aria -i <input_file> -opt <ROL_inputs> --beta
```

where the input file is the target Aria input deck, ROL inputs is an xml file containing inputs to ROL, and the --beta flag is required. Among the inputs contained in the xml file is the location of the experimental data file, optimization algorithm inputs, and termination criteria. Example ROL input files can be provided upon request.

To build Inverse Aria, compile with the following command (e.g. for the release build)

```
$  bake InverseAria -e release
```

An adjoint source term must always be present on all blocks when using adjoint-based gradients. This is not required for optimization with finite difference gradients such as the Arrhenius source terms. This term will be zero on the forward solves, and it is filled with the derivative of the reduced objective function with respect to temperature during the adjoint solves. The adjoint source term is specified as follows:

```
Source for energy on all_blocks = Optimization value = 0
```

Input 3.2.1. Adjoint source term syntax

A subset of boundary conditions are supported by Inverse Aria. These boundary conditions are no flux (default), Dirichlet, flux, generalized convection, and generalized radiation. Usage of boundary conditions outside of these will result in unexpected behavior in the adjoint solve. The optimization keyword is no longer required for specifying boundary conditions.

Two output files are produced in addition to the normal output from Aria simulations that allow the user to track the optimizer's progress. "ROL_Messages.txt" displays optimizer convergence values at each iteration such as the objective and gradient norms. "DesignVariableIterations.txt" summarizes the design variable values at each iteration, where the values are tagged by material or surface names. In the case of heterogeneous material property inversion, the minimum and maximum values are reported with the material name.

3.2.3. Optimization .xml Inputs for Inverse Aria

In addition to the required parameters for a ROL optimization problem, there are a few Inverse Aria specific inputs in the ".xml" optimization input file. First, the optimization data file is always required. It consites of a text file with rows corresponding to individual measurement locations where the first entry in the row is the node number on the mesh followed by a time history of experimental measurements separated by commas. The syntax for pointing Inverse Aria to this file is shown in Input 3.2.2.

```
<ParameterList name="Problem Data">
  <Parameter name="Data File" type="string" value="data_file.txt" />
</ParameterList>
```

Input 3.2.2. Problem data file

Regularization methods can improve the stability of the inverse problem. Currently, Tikhonov regularization is available in Inverse Aria, and it is activated by adding an "Inverse Aria" parameter list with the regularization weight(s) to the ".xml" optimization input file. Examples are shown for a single design variable problem and multi-design variable problem by Inputs 3.2.3 and 3.2.4 respectively.

```
<ParameterList name="Inverse Aria">
  <Parameter name="tikhonovParameter" type="double" value="500.0" />
</ParameterList>
```

Input 3.2.3. Single design problem regularization input

```
<ParameterList name="Inverse Aria">
  <ParameterList name="tikhonovWeights">
    <Parameter name="Material" type="double" value="2.0" />
    <Parameter name="Flux" type="double" value="10.0" />
    <Parameter name="GapConductance" type="double" value="5.0" />
  </ParameterList>
</ParameterList>
```

Input 3.2.4. Multi-design problem regularization input

3.3. Inverse Problems

The chapter covers the current capabilities of Inverse Aria with examples of the required inputs. The problems covered are thermal conductivity inversion, steady boundary heat flux inversion, contact resistance inversion, and Arrhenius source term inversion with finite difference gradients. The input decks and meshes for the examples can be found at docs/fused/InverseAria/Examples.

3.4. Thermal Conductivity

An example use case for thermal conductivity inversion can arise in situations where two materials are joined together by some sort of adhesive and the quality of the bond is unknown. A simplified example is shown in Figure 3-1, where two cylinders are joined by a thin material of unknown thermal conductivity. The domain is modeled as 2D axi-symmetric with a heat flux applied to the outside of the outer cylinder and temperature measurements are only available at this outer surface. In this example, synthetic data at each surface node were generated from an Aria simulation with specified thermal conductivities in the joining layer.

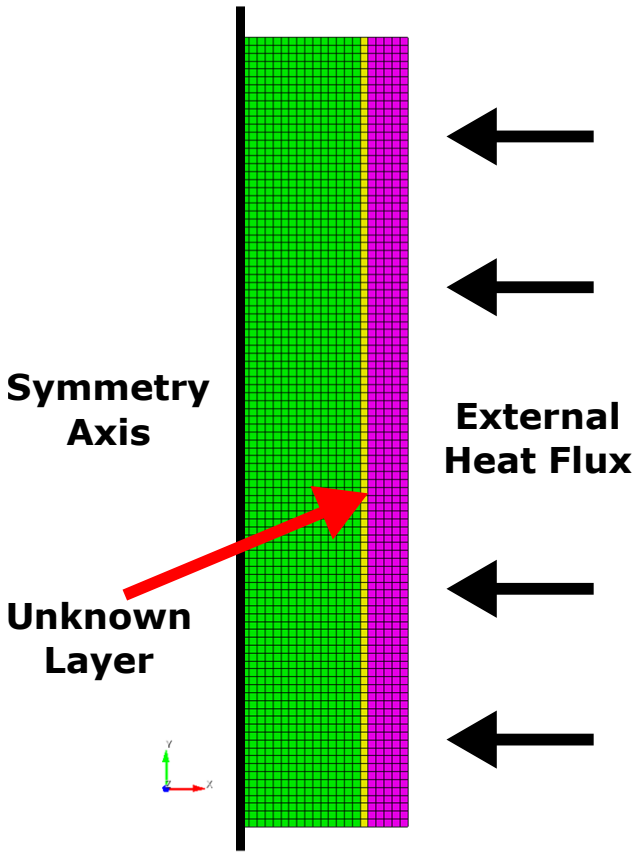


Figure 3-1. – Domain of the example thermal conductivity inverse problem.

There are two options in the material specification when inverting for thermal conductivity. Choosing homogeneous will result in inverting for a single thermal conductivity value for the entire material, whereas using heterogeneous will invert for a thermal conductivity value for every element with that material. the syntax for a homogeneous thermal conductivity inversion problem is

```
Begin Aria Material MatA
    Density              = Constant rho = 2702.
    Specific Heat        = Constant cp  = 903.
    Thermal Conductivity = Optimization type = homogeneous k = 300
    Heat Conduction      = Basic
End   Aria Material MatA
```

Input 3.4.1. Homogeneous thermal conductivity inversion

and a heterogeneous problem is defined by

```
Begin Aria Material MatA
    Density              = Constant rho = 2702.
    Specific Heat        = Constant cp  = 903.
```

```
    Thermal Conductivity = Optimization type = heterogeneous k = 300
    Heat Conduction      = Basic
End   Aria Material MatA
```

Input 3.4.2. Heterogeneous thermal conductivity inversion

The user has the option to provide lower and upper bounds for the design variable on the thermal conductivity line with the keywords `lower_bound` and `upper_bound` and values separated by equals signs.

A simple example can be found at Examples/Thermal_Conductivity/Homogeneous_1_Block. The key files are:

- `layered_2D_inv.i` - Inverse Aria input deck
- `inverseInput.xml` - ROL input parameters
- `layered_2D_data.txt` - Synthetic temperature data
- `layered_2D.i` - Aria input deck for generating synthetic data
- `layered_2D.jou/g` - Mesh journal file and genesis file

In this example, a 500 kW/m^2 heat flux is applied to the outside of the cylinder, and the thermal conductivity of the joining layer is treated as homogeneous. The inverse solution is found quickly for this simple problem as shown by the objective function and gradient norm in Figure 3-2.

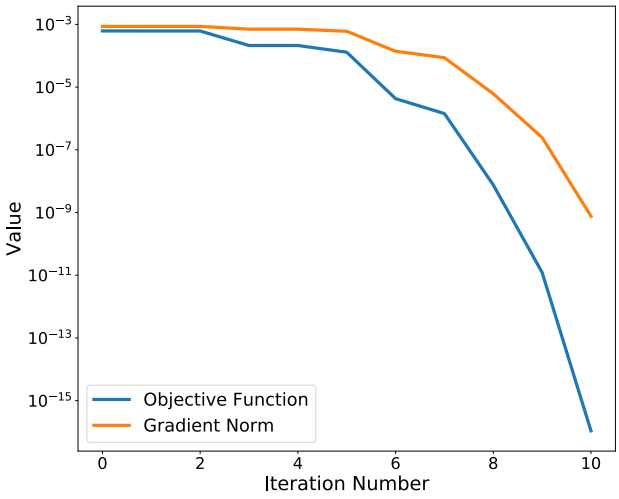


Figure 3-2. – Objective function and gradient norm at each iteration of the optimizer.

3.5. Steady Boundary Heat Flux

To invert for a steady heat flux on a surface, the “Optimization_Design” keyword is used.

```
BC Flux for Energy on surface_2 = Optimization_Design value = -1e6
```

Input 3.5.1. Steady heat flux inversion

The example problem is composed of three materials with different material properties and a steady flux on the right half of the top surface as shown in Figure 3-3 (left). Material A is a conductive material, and materials B and C represent different internal layers with varying material properties. In real experiments, data may only be available on one surface of the test article, and for this example synthetic data are generated at the nodes along the bottom surface.

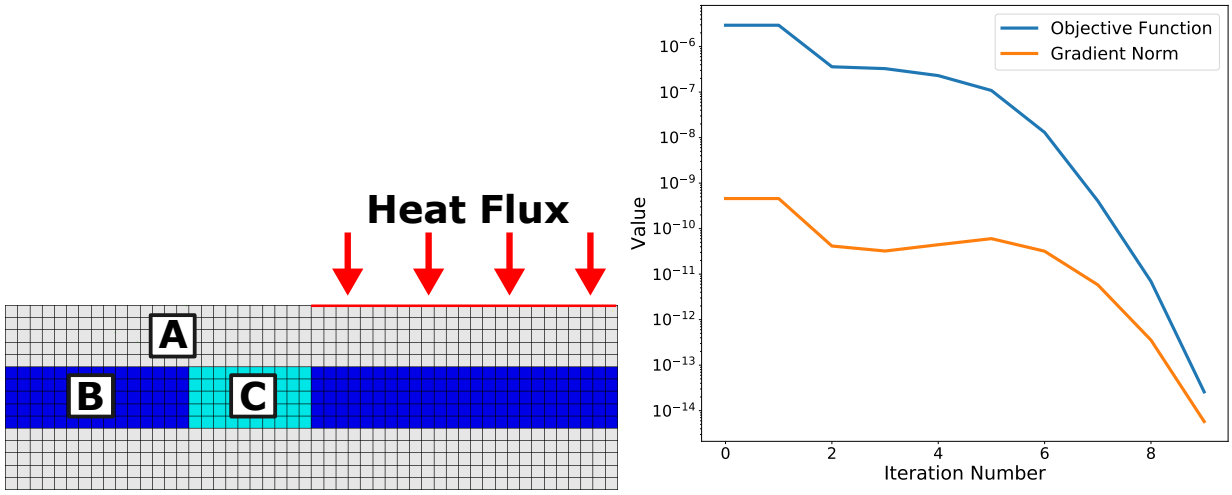


Figure 3-3. – Domain of the example heat flux inverse problem (left) and residuals for the inverse problem (right).

For the inverse problem we imagine a scenario where the heat flux along the entire top surface is unknown, and we search for the fluxes on the left and right halves of the top surface (i.e. two design variables). In the example input deck (Examples/Heat_Flux/Steady/layered_2D_inv.i), we set an initial guess of 5 kW/m^2 on both halves and the optimizer quickly finds the solution (Fig. 3-3 right) of 30 kW/m^2 on the top right half of the domain. The values of the heat fluxes at the top surface can be viewed in the heartbeat file: `layered_2D_flux_inv.txt`.

3.6. Transient Boundary Heat Flux

Reconstructing an unknown transient heat flux can be accomplished in a variety of ways that all require the specification of a functional dependence of the heat flux on time. In the present formulation, the heat flux is modeled as a piecewise linear function between a set of user-specified times. Two options are available for specifying these times: constant time intervals and arbitrary time points.

To use constant time intervals, the “num_times” keyword is added to the boundary condition specification.

```
BC Flux for Energy on surface_2 = Optimization_Design $
    value = -1e6 num_times = 2
```

Input 3.6.1. Transient heat flux inversion with constant time intervals

In this example input line, the total simulation time is divided in to two equal intervals with three design variables (unknown heat fluxes) located at time 0, one half the simulation time, and the final simulation time. For example, if the total simulation time is 300 seconds and the user selects "num_times" = 2, the transient heat flux will be reconstructed at 0 s, 150 s, and 300 s with piecewise linear functions between each of these points. The initial guess for the optimization problem is a constant heat flux specified with the "value" keyword.

If the user desires more control over the specific time points and initial guess for the heat flux, they can use a tabular user function. This input syntax will be familiar to current Aria users, as it is commonly used to specify time varying boundary conditions or temperature dependent material properties. The syntax for the boundary condition line is as follows

```
BC Flux for Energy on surface_2 = Optimization_Design_User_Function $
    NAME = input_flux X = time
```

Input 3.6.2. Transient heat flux inversion with time intervals specified by a user function

This requires the specification of a user function in the Sierra domain of the input file. The name of this function must match the name in the boundary condition line ("input_flux" in this case).

```
BEGIN DEFINITION FOR FUNCTION input_flux
    type is piecewise linear
    begin values
        # t(s),     W/m2
        0           -2.0e4
        200         -1.0e5
        300         -5.0e4
    end values
END DEFINITION FOR FUNCTION
```

Input 3.6.3. Transient heat flux inversion user function example

In the function above, the optimizer will invert for the heat flux at times 0 s, 200 s, and 300 s using the values provided in the table as the initial guess.

An example transient heat flux inverse problem was created based on the steady heat flux simulation in Section 3.5. This example uses the same geometry, heat flux location, and temperature measurement locations on the bottom of the domain as the steady heat flux example.

Synthetic temperature data is generated using a heat flux profile that begins at 50 kW/m^2 and remains constant for 60 seconds (Fig. 3-4 left). The heat flux then decreases linearly to 0 kW/m^2 over the next 40 seconds and remains at zero until the end of the simulation time.

The time interval for the inverse solution of the heat flux was set to 20 seconds, and the initial guess was set at 10 kW/m^2 for 100 seconds decreasing to 0 kW/m^2 at 120 seconds as seen in Figure 3-4 left. It is important to note that the time intervals are informed by the heat conduction time between the heated surface and the measurement locations. This heat conduction time is modeled as $t = \delta^2/\alpha$, where δ is the distance between the heat must travel and α is the thermal diffusivity of the material. For the thermally "closest" thermocouple to the surface in this geometry, the heat conduction time is approximately 9 seconds. This is effectively a lower limit on the fidelity of the transient reconstruction as there is insufficient information to resolve smaller times.

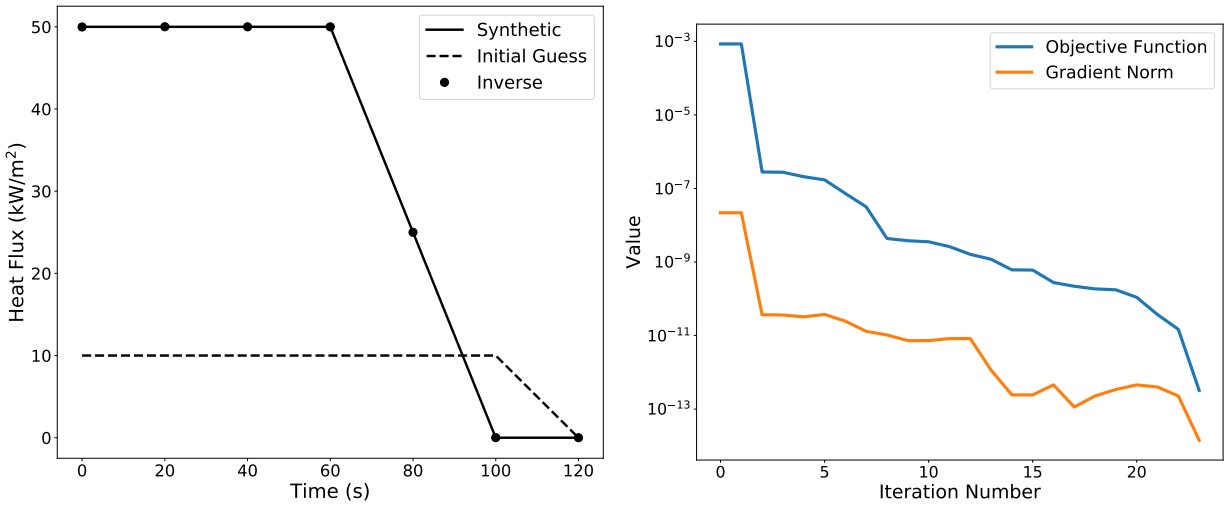


Figure 3-4. – Transient heat flux with inverse solution (left) and residuals for the inverse problem (right).

Figure 3-4 right shows the objective function and norm of the gradient versus optimization iteration. Note that after a large initial decrease in the objective function and gradient, several iterations are required to reach the "true" synthetic solution due to the loss of thermal information as heat diffuses from the heated surface to the temperature measurement locations.

3.7. Thermal Contact Resistance

Thermal contact resistance is used to model is used to model resistance to heat transfer across a gap between two materials in contact. This interaction is a function of multiple properties such as material hardness, surface finish, interstitial material, etc. The reader is directed to the Aria User Guide for a detailed description of the numerical implementation of this model [23].

Currently, three enforcement options for thermal contact resistance are available in Aria: `CONDUCTANCE`, `CONTACT_RESISTANCE`, and `GAP_CONDUCTANCE`. Of these, `CONDUCTANCE` and

CONTACT_RESISTANCE are supported by Inverse Aria. The design variable is specified by the gap conductance coefficient, which is the inverse of the contact resistance. This is done in the enforcement block of the contact definition in the Aria input file with the "Optimization" keyword as follows, where the "value" is set to the initial guess for the optimization problem.

```
Begin Enforcement enf_1
    Enforcement for Energy = Conductance
    Gap Conductance Coefficient = Optimization value = 100
End   Enforcement enf_1
```

Input 3.7.1. Gap conductance coefficient design variable specification

An example is provided in Examples/Contact_Resistance. The domain consists of an 2D axi-symmetric cylinder where two materials are in contact, but the contact at different sections of the material interface is uncertain. One could imagine a scenario where two materials are bound together similar to the example in Section 3.4, where the interface is instead represented by thermal contact boundaries. In this example, we choose three surfaces to have unknown contact as highlighted in Figure 3-5 (left), with a heat flux applied to the outside of the cylinder and temperature measurements along the central axis and outside surface. We generate synthetic temperature data with Aria and defined thermal contact resistances and use this to solve the inverse problem with an initial guess of 100 W/m^2/K for the conductance at each interface. Inverse quickly finds the synthetic conductance values in 9 iterations as shown in Figure 3-5 (right).

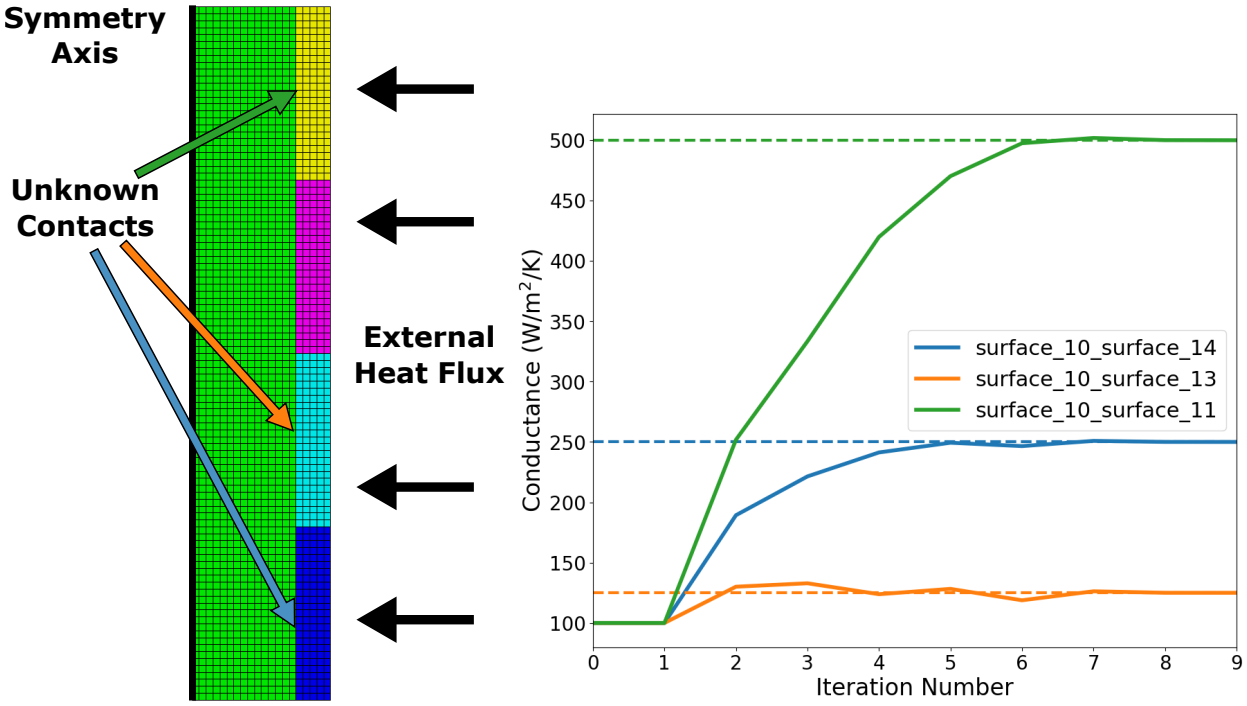


Figure 3-5. – Domain of the example contact resistance inverse problem (left). Contact arrow colors correspond to line colors in the plot of design variable progress at each optimization iteration (right). Dashed lines indicate the "true" values used to generate synthetic temperature data.

3.8. Arrhenius Source Terms with Finite Differences

The reaction source term inversion feature differs from the other inverse problems in that it uses finite differences and not adjoint solves to compute gradients of the objective function. This requires less modification of the input file for inverse problems, but the inverse solution suffers from the lack of adjoint-based gradients. This feature has been primarily implemented as a stepping-stone towards adjoint solves with reaction source terms in Inverse Aria.

The Arrhenius form of a reaction is given as

$$k = Ae^{-E_a/RT} \tag{3.8.1}$$

where the rate constant (k) is a function of the pre-exponential factor (A), activation energy (E_a), ideal gas constant (R), and temperature (T). Typically, calibration of reaction models involves searching for an A and E_a that fit experimental calorimetry data. Other parameters such as the heat of reaction and concentration function are also candidates for inversion.

Inverse Aria currently supports inversion for A and E_a for an arbitrary number of reactions using the following syntax in the General Chemistry block.

```
Begin Optimization Reaction_Name
    Optimize = A log_transform = true
    Optimize = Ea lower_bound = 1 upper_bound = 1e10
End
```

Input 3.8.1. Reaction source term inversion

In this example syntax, `Reaction_Name` must match the name of the reaction block with unknown parameters. Initial guesses are set in the reaction block and the rate function must be Arrhenius. The user can specify A and/or E_a as design variables with the `Optimize =` command. Providing lower and upper bounds on the potential values for the design variable are optional. Log transforming the design variable before sending it to ROL is also optional, with the default being false.

Unlike problems that use adjoint-based gradients, solving a problem with finite differences does not require modification of the boundary condition specification or the addition of an adjoint source term in the input file. Only the optimization block must be provided by the user in the General Chemistry block of the Aria input file. Additionally, the ROL input file must have the following line to enable finite difference gradients

```
<Parameter name="Use FD Gradient" type="bool" value="true" />
```

Input 3.8.2. Switching to finite difference gradients in the ROL input file

The user is directed to the regression test suite (InverseAria_rtest) for examples with one and two reactions (inverseProblems/one_rxn_one_param and inverseProblems/two_rxn_two_param).

4. OPTIMAL EXPERIMENTAL DESIGN

4.1. Introduction to InverseOED

Inverse Optimal Experiment Design (InverseOED) is a massively parallel, nightly tested, physics-agnostic Sierra app that rapidly optimizes sensor locations in order to minimize targeted forms of uncertainty in the types of experiments used to estimate hidden parameters. In other words, InverseOED performs sensor placement optimization for inverse problems.

Inverse problems and experiments are inherently coupled since it is often difficult to directly measure the required quantities of interest and because the measured data is restricted to a small subset of the spatial domain. Many experiments aim to estimate unknown parameters or boundary conditions from a finite set of discrete measurements using an inverse problem. For example, in acoustics vibration tests, the loudspeaker inputs that are needed to excite a structure to emulate a flight environment are estimated via a frequency domain inverse problem.

InverseOED is based on the classical OED theory for linear inverse problems. The theory includes additive noise models and uses the covariance of the estimated model parameters to quantify the suitability of an experiment design. In general, an optimal design is a solution to the optimization problem

$$q^* \in \underset{q}{\operatorname{argmin}} \ \Psi(C(q)), \tag{4.1.1}$$

where Ψ is a scalar function acting on matrices, C is the covariance matrix associated with the estimated parameters, and q are candidate sensor weights. One can interpret the optimal design as the design that minimizes uncertainty in the estimated parameters. An equivalent interpretation is that the optimal design maximizes sensor information.

InverseOED provides a variety of objective functions and options so that the optimal design is catered to the experimenter's needs. For example, InverseOED provides different scalar functions Ψ, called optimality criteria, that result in different measures of uncertainty such as average prediction variance, maximum prediction variance, and average parameter variance. There are also options for controlling the robustness of an experiment design using risk-adapted optimality criteria. Finally, InverseOED also provides mechanisms for enforcing total sensor budgets, enforcing fixed sensor locations, and placing multi-measurement devices such as triaxial accelerometers.

4.2. Input Deck Introduction

The input deck is an XML formatted file (**.xml**) that is used to specify the optimality criterion, the transfer matrix, and the parameters of the optimization algorithm. This section explains the input deck parameters needed to run InverseOED.

XML files have parameter lists and parameters. The syntax for a parameter list and parameter are given below.

```
〈 ParameterList name="List Name"/〉
  〈 Parameter name="name" type="string/int/bool/double" value="value" 〉
〈/ ParameterList〉
```

For the remainder of the document, we will remove the angular brackets and the type for convenience and space.

All input decks must have a **ParameterList name = "Inputs"** at the beginning that encompasses all the parameters the user wishes to pass to the algorithm.

```
ParameterList name = "Inputs"
     User provided parameters
ParameterList
```

4.3. ParameterList: OED

The solution of the OED depends on the optimality criterion selected. This is specified with the **Optimality Type** parameter within the **OED** parameter list as shown below.

```
Input Deck: OED
ParameterList name="OED"
     Parameter name="Optimality Type"  value="A/B/C/D/I/R/E"
ParameterList
```

For the gradient-based optimization algorithm, the user can choose between **A, C, D, I** and **R**. **D** and **E** are available for the greedy algorithm, which is discussed in greater detail in section 4.6. Table 4-1 provides a brief description for each criterion. The theory documentation describes each optimality criterion in greater detail.

Criterion	$\Psi(C(p))$	Description
A	$Tr(C)$	Average estimation variance (MSE)
C	$v^T C v$	Variance of $v \in \Re^m$ times the estimator
D	$det(C)$	Volume of the uncertainty ellipsoid
I	$\mathbb{E}[h^T C h]$	Average prediction variance (MSPE)
R	$AVaR_\beta[h^T C h]$	Tail average prediction variance for $\beta \in [0,\ 1]$
E	$max(\lambda(C))$	Maximum eigenvalue

Table 4-1. – Optimality Criteria: $C \in \Re^{mxm}$ is the covariance matrix of the design parameters.

4.3.1. Initial Design

By default, the initial design weights are equal across all candidate sensors. If desired, the user can specify the initial design weights, which may be useful if the user wants to warm start the algorithm at a set of specified weights or if an optimization run terminated early and a restart is needed.

- **Uniform Initial Guess**: If true, then the initial weights at each sensor are equal. If false, then an initial design guess must be provided. Default value is true.
- **Initial Design Guess**: A user specified text file containing the initial weights on each sensor. This file will have the same format as a resulting optimal design text file. See section 4.5 for a description of the results file format. This option can be used to restart an optimization problem if the previous optimization algorithm terminated early. Note that the sum of the weights should equal one.

```
Input Deck: Initial Design
ParameterList name="OED"
     Parameter name="Uniform Initial Guess"  value="true"
     Parameter name="Initial Design Guess" value="initial.txt"
ParameterList
```

4.3.2. Baseline sensors

The user can specify a set of sensors that must be included in the final sensor design. This option is available for both the greedy and Rapid Optimization Library's (ROL) gradient-based OED algorithms. In terms of the algorithm, the transfer matrix degrees of freedom associated with the fixed sensor set is always included in the covariance calculation. The algorithm optimizes only the excluded set of sensors. To enable this feature, the user needs to specify a list of indices corresponding to the rows of the transfer matrix via a text file. This file must begin with the total number of baseline sensors followed by the indices. These indices are written for C++, so 0 indicates the first sensor. The list is provided to the xml file via the following parameter sublist:

```
Input Deck: Baseline Sensors
ParameterList name = "OED"
     Parameter name = "Use Baseline Sensors" value = "true"
     Parameter name = "Baseline Sensors File" value = "baseline.txt"
ParameterList
```

The following example baseline sensor text file includes three total sensors: the fourth (3), first (0), and seventh (6) sensor.

```
Example Baseline Sensor File
3 // Total number of sensors
3
0
6
```

4.4. ParameterList: Linear Model

Currently, optimal experiment design assumes a linear regression model. In other words, there exists a matrix $H(q) \in \mathbb{C}^{n \times m}$ that maps the design parameters of interest $\theta(q) \in \mathbb{C}^m$ to the measured response data $y(q) \in \mathbb{C}^n$, where m represents the number of design parameters and n represents the number of measurement locations. We refer to the linear operator H as a transfer matrix. The modal matrix used for modal expansion, the frequency response function for frequency domain control problems, or the convolution matrix for time domain control problems are a few common examples of the transfer matrix.

The simplest case is where each row of the transfer matrix corresponds to a single candidate sensor and correspondingly a sensor weight. The more complex case is when there are multiple observations of the data at a single sensor location. For example, in the frequency domain the number of observations is equal to the number of frequency lines.

The transfer matrix is provided in the **Linear Model** parameter list. There are three options for parsing the transfer matrix based on the domain type.

- **General**: Any transfer matrix can be specified using the general framework
- **Frequency Domain**: Specific to the frequency domain where the response at each frequency line is independent
- **Time Domain**: Specific to the time domain where the response at each time step is dependent

The following sections describes each domain type and the associated input deck.

4.4.1. General Framework

The user must write the transfer matrix to a *.txt* file. In the general framework, the user provides a transfer matrix text file, a transfer matrix dimension file, and specifies the number of models where models indicates the number of observations per sensor.

```
Input Deck: General Domain
ParameterList name="Linear Model"
     Parameter name="Domain Type" value="General"
     Parameter name="Number of Models"  value="2"
     Parameter name = "Transfer Matrix"  value = "transferMatrix.txt"
     Parameter name="Transfer Matrix Dimension" value = "tmDimFile.txt"
ParameterList
```

Further details on each parameter:

- **Transfer Matrix**: User provided transfer matrix text (.txt) file that contains the full transfer matrix.
- **Transfer Matrix Dimension**: User provided transfer matrix dimension text (.txt) file. The first and second entry are respectively the total number of rows and columns of the full transfer matrix.
- **Number of Models**: Total number of possible measurements at a sensor. For a frequency domain problem, this value is the total number of frequencies, and in the time domain, the total number of time steps. The user should check that the total number of rows of the transfer matrix divided by the specified Number of Models is equal to the number of possible sensors.

For example, consider a frequency domain problem with real valued (no-imaginary part) transfer matrices. Let n_0 be the number of frequency lines. The full transfer matrix will have $n * n_o$ rows and $m * n_o$ columns. The format of the transfer matrix is as follows:

$$\mathbf{H} = \begin{pmatrix} H(\omega_1) & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & H(\omega_2) & 0 & \cdot & \cdot & 0 \\ \cdot & 0 & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & 0 \\ 0 & 0 & \cdot & \cdot & 0 & H(\omega_{n_o}) \end{pmatrix}$$

where $H(\omega_i)$ is the frequency response function (FRF) at frequency ω_i. Again $H(\omega_i)$ has n rows for the n sensor locations, and m columns for the number of design parameters associated with the i^{th} frequency. The rows and columns of $H(\omega_i)$ must be ordered consistently for each frequency, and the number of sensors and number of parameters must be equal for all frequencies. Note that the full transfer matrix $\mathbf{H}$ is block diagonal since the measured response at ω_i is independent of

parameters of ω_j for $i \neq j$. This format becomes more complex if $H(\omega_i)$ is complexed value. In that case, $H(\omega_i)$ contains both the real and imaginary parts in the following block form.

$$\mathbf{H} = \begin{pmatrix} \mathbf{H}_R & -\mathbf{H}_I \\ \mathbf{H}_I & \mathbf{H}_R \end{pmatrix}$$

where $\mathbf{H}_R$ is the full transfer matrix containing the real part, and $\mathbf{H}_I$ contains the imaginary part.

4.4.2. Frequency Domain

To avoid the parsing confusion for a complex-valued frequency domain transfer matrix, we provide a simpler parsing interface where the user only needs to specify the real part of the transfer matrix and the imaginary part as two separate files in the following format:

$$\mathbf{H} = \begin{pmatrix} H(\omega_1) \\ H(\omega_2) \\ . \\ . \\ . \\ H(\omega_{n_o}) \end{pmatrix}$$

Instead of providing a transfer matrix dimension file, the dimensions of the transfer matrix are provided in the **Linear Model** parameter list as shown below. If there is no imaginary component then set the **Complex Valued** parameter to false. The user can also use the frequency domain parser to parse a transfer matrix that has only a single observation, such as the case of the modal matrix, by setting the **Number of Frequencies** equal to one.

Input Deck: Frequency Domain

```
ParameterList name = "Linear Model"
 Parameter name = "Domain Type" value = "Frequency"
 Parameter name = "Number of Parameters" value = "3"
 Parameter name = "Number of Outputs" value = "10"

 ParameterList name = "Frequency Domain"
  Parameter name = "Complex Valued" value = "true"
  Parameter name = "Number of Frequencies" value = "2"
  Parameter name = "Real Frequency Response Function" value = "realf"
  Parameter name = "Imaginary Frequency Response Function" value = "imagf"
 ParameterList
ParameterList
```

4.4.3. Time Domain

The third parsing option is for time domain type problems. When the transfer matrix is a function of time, the user specifies the impulse response for each parameter-output combination. The Domain Type option should be set to Time. By doing so, the code internally converts the impulse responses into a convolution matrix which takes a Hankel matrix form. The number of models will be equal to the total number of time steps. The time domain transfer matrix takes the following form:

$$\mathbf{H} = \begin{pmatrix} H(t_1) \\ H(t_2) \\ . \\ . \\ . \\ H(t_{n_o}) \end{pmatrix}$$

where $H(t_i) \in \Re^{nxm}$ is the impulse response at the i^{th} time step between the m parameters and n outputs (the measurement locations). For example, the first row of $H(t_i)$ is the response at $t = i$ at outputs $1...m$ due to a delta function at the first parameter.

```
Input Deck: Time Domain
ParameterList name = "Linear Model"
     Parameter name = "Domain Type" value = "Time"
     Parameter name = "Number of Models" value = "2"
     Parameter name = "Transfer Matrix" value = "matrix.txt"
     Parameter name = "Transfer Matrix Dimension" value = "dimFile.txt"
ParameterList
```

Time domain inverse problems typically have hundreds of thousands of time steps, which appear as observations in the OED app. The current version of the InverseOED app struggles with a large number of observations due to the significant computational demand. We typically do not recommend solving time domain OED problems with this app.

> Time domain problems may be too computationally costly for the OED app.

4.4.3.1. Multi-axis sensor placement (Triaxial sensor placement)

InverseOED is capable of optimizing multi-axis sensors such as triaxial accelerometers. Specifying triaxial sensors is described in the Greedy algorithm section. See Sections 4.6.1 or 4.6.2 for details.

4.4.4. Robustness to Sensor Dropout

InverseOED also employs a gradient-based approach to determine optimal sensor locations that are robust to sensor dropout or missing data. Sensor dropout parameters are specified under the **OED.Robust** parameter sublist. The following is an example of the input deck configuration for specifying sensor dropout parameters:

```
Input Deck: Sensor Dropout
ParameterList name = "OED"
     ParameterList name = "Robust"
     Parameter name = "Use Sensor Dropout" value = "true"
     Parameter name = "Dropout Type" value = "User-Defined/Bernoulli"
     Parameter name = "Number of Dropout Samples" value = "3"
     Parameter name = "Dropout Values" value = "pof.txt"
     ParameterList
ParameterList
```

Using **Bernoulli** dropout type, the sensor dropout is modeled by a Bernoulli random variable. The Bernoulli random variable returns a one or zero with a specified probability indicating whether the sensor has or not. The user must provide a list of probabilities in a text file specified in **Dropout Values**. The **Dropout Values** file should contain a column of probabilities of failure, where row i corresponds to the probability of failure of sensor i. Each value must be between 0 and 1, with 1.0 indicating a 100% chance of dropout (i.e., guaranteed sensor failure). The number of rows in the file should match the number of candidate sensors. The **Number of Dropout Samples** parameter indicates how many draws from the Bernoulli random variable should be taken. Decreasing the number of dropout samples reduces computational runtime but at the expense of design robustness. For example, if there are three candidate sensors, the **Dropout Values** might look like this:

```
Example of Dropout File
     0.1 // Low probability of failure
     0.9 // High probability of failure
     0.5 // 50-50 chance of failure
```

Using the **User-Defined** dropout type, the user provides specific samples of sensor probability of failures. Each column in the **Dropout Values** represents a dropout sample. The number of dropout samples specified in **Number of Dropout Samples** must equal the number of columns in the file. A value of 0.0 indicates a zero POF and hence that the sensor is included in the design calculation.

For example, if there are three candidate sensors and the goal is to ensure robustness to a single sensor failing, the **Dropout Values** might look like this:

```
Example of Dropout File
     1 0 0
     0 1 0
```

```
      0 0 1
```

where column one (1, 0, 0) indicates that the first sensor has failed, column two (0, 1, 0) indicates that the second sensor has failed, and column three (0, 0, 1) indicates the third sensor has failed.

We recommend running a gradient check before attempting to solve the optimization problem,

```
Input Deck: Gradient Check
Parameter name = "Check Optimization Problem" value = "true"
```

There is a chance that the covariance matrix becomes rank deficient if the number of failed sensors is large. In this scenario, the optimization algorithm may become unstable or fail completely. A passing gradient check will indicate an adequate number of sensors.

4.5. Executing InverseOED and Results

4.5.1. InverseOED executable

To execute the InverseOED app, type the following in the command line:

```
mpirun    -np    numProcs    oed_inverse    -opt    inputXML.xml
```

NumProcs is the number of processors used for parallel runs and **inputXML**.xml is the xml file which contains the user specified options.

4.5.2. Parallel Runs

InverseOED splits the number of processors between algebraic operations for design variables (sensor weights) and the stochastic objective function evaluations. InverseOED automatically selects a reasonable processor configuration for each type of OED problem. In some cases, the user can improve the algorithm's performance by playing around with the **Number of Design Processors** value, however the default tends to render sufficient performance.

The user specifies which fraction of processors are devoted to algebraic operations as follows:

```
Input Deck: Processors
ParameterList name = "Problem"
     ParameterList name = "Design"
          Parameter name = "Number of Design Processors" value = "4"
     ParameterList
ParameterList
```

For example, if *Number of Design Processors* is set to 5, and there are 60 processors, then 12 processors are devoted to the objective function and each of the 12 has 5 processors devoted to

algebraic operations. We recommend that *Number of Design Processors* roughly be 10 percent of the total number of processors.

4.5.3. Results

After the optimization algorithm converges 3 (4 for I optimality) text files are outputted to the current directory. *I_optimal_design_final.txt* contains the solution to the OED on the first processor. The first column corresponds to the sensor label. This label is ordered consistently with the order in which sensors appear in the transfer matrix. Columns 2 and 3 can be ignored. Column 4 represents the probability measure of that sensor location. For a fully converged solution where the optimization constraints are met, this value is between 0 and 1. The sum of all probabilities equals 1. The probability measures can be interpreted as the percentage that the sensor should be experimentally sampled. The candidate sensors with weights that are non-zero should be selected for the experiment design.

If InverseOED is ran in parallel then **numProcs** total number of *I_optimal_design* files are printed and concatenated into *I_optimal_design_final*.

The user can also specify the tail of the file outputs generated from the greedy algorithm as shown below.

```
Input Deck:Output File Tail
ParameterList name = "Inputs"
     Parameter name = "Output File Tail" value = "userDefinedTail"
ParameterList
```

4.6. Greedy Algorithm

The default OED algorithm in InverseOED is ROL's convex optimization algorithm. This algorithm may not be suitable if the user needs to constrain the final design by a total sensor budget. In order to enforce a total sensor budget, InverseOED offers a greedy optimization algorithm that strictly enforces the sensor budget. Greedy optimization is a heuristic approach that does not guarantee the solver reaches the global optimal solution. However, the optimality criteria that are available for Greedy are submodular or near-submodular, which guarantees that the greedy solution will be close to optimal.

The greedy algorithm is available for the D-criterion and E-criterion. The E-criterion is the maximum eigenvalue of the covariance matrix and is only available for the greedy algorithm. In order to run the greedy algorithm, the user provides the following parameters to the input deck:

```
Input Deck: Greedy
ParameterList name = "Inputs"
      Parameter name = "Use Greedy Method" value = "true"
ParameterList
```

The total sensor budget is specified in the greedy parameter list as follows:

```
Input Deck: Greedy
ParameterList name = "Greedy"
     Parameter name = "Budget"  value = "5"
ParameterList
```

4.6.1. *Multi-axis sensor placement (Original Version)*

> There are two ways to specify multi-axis sensor placement. This section is the original method, maintained for backwards compatability. However, we recommend the new version, section 4.6.2, which also support mutliple sensor types and multiple budgets.

The greedy algorithm in InverseOED is capable of optimizing multi-axis sensors such as triaxial accelerometers.

The user provides a map between the rows of the transfer matrix and an unique sensor label. The first row is the total number of sensors. The number of rows (not including the first row) should be equal to the total number of rows of the transfer matrix. Here is an example of a triaxial sensor mapping (3 degrees of freedom (DoFs) per unique sensor):

```
Example Transfer Map File
3 // Total number of sensors
0
0
0
1
1
1
2
2
2
```

In the above example, there are 3 total sensors where DoFs 1–3 correspond to sensor 1, DoFs 4–6 correspond to sensor 2, and DoFs 7–9 correspond to sensor 3. Finally, the user specifies the multi-axis option and the transfer map file in the xml under the **Linear Model** sublist as shown below

```
Input Deck: Multi-axis Sensors
ParameterList name = "Linear Model"
     Parameter name = "Use Transfer Map" value = "true"
     Parameter name = "Transfer Map File" value = "transferMap.txt"
ParameterList
```

4.6.2. Multiple Budgets and Multiple Sensor Types

The greedy algorithm supports the optimization of multiple sensor types. For example, this method can optimize triaxial and uniaxial sensors simultaneously. The method supports either one total sensor budget or multiple sensor budgets: one budget for each sensor type. If one budget is specified, the algorithm optimizes all sensor types until the total sensor budget is satisified. If multiple budgets are specified, then the algorithm runs until each sensor budget is satisified.

We will use the following keywords throughout this section of the user manual:

- *Sensor Label*: An integer representing the sensor type. For instance, uniaxial sensors are labeled as 0, and triaxial sensors as 1.
- *Sensor ID*: A unique integer assigned to each sensor, ranging from 0 to N, where N is the total number of candidate sensors.
- *Transfer Indices*: The rows of the transfer matrix to which a sensor is mapped. For example, a triaxial sensor might correspond to indices 10, 11, 12, while a uniaxial sensor might map to index 10.

In summary, the user needs to provide the following information in order to solve multiple sensor budget problems:

- A unique sensor label that identifies the type of sensor. This can be done via the **Sensor Labels** xml parameter.
- For each candidate sensor, a unique sensor ID and its mapping to the transfer matrix indices. This can be specified using the **Transfer Map File** xml parameter
- A budget for each unique sensor label. These budgets should be provided as a list using the **Budgets** xml parameter list.

Multi-Budget: Sensor Label Specification

When using multiple budgets, a sensor labels file must be provided in the Greedy xml sublist as shown below.

```
Input Deck: Sensor Labels
     ParameterList name = "Greedy"
          Parameter name = "Sensor Labels" value = "sensor_labels.txt"
     ParameterList
```

The first line of the sensor labels file specifies the total count of candidate sensors. Following this, each line contains a sensor label, indicating the sensor type, with each line corresponding to a unique sensor ID. In the provided example, there are three triaxial sensors labeled as 0 and two uniaxial sensors labeled as 1, making a total of five candidate sensors.

The sensor labels file structure is as follows.

```
Example Sensor Labels File
     5 // Total Candidate Sensors
     0 // ID = 0, Label = 0 (Tri)
     0 // ID = 1, Label = 0 (Tri)
     0 // ID = 2, Label = 0 (Tri)
     1 // ID = 3, Label = 1 (Uni)
     1 // ID = 4, Label = 1 (Uni)
```

It is important to note that sensor labels cannot be arbitrary integers. The first sensor type must be zero, and each subsequent sensor type should be assigned the next consecutive integer value (e.g., one, two, three, and so on).

Multi-Budget: *Transfer Map Specification*

The transfer map file links each sensor ID to specific rows in the transfer matrix (indices). The original transfer map format is still compatible for single sensor/budget scenarios (see section 4.6.1). However, the following new format must be used for multiple budget problems. In addition, we recommend this new format for single budget problems as well.

The new transfer map file format is as follows. The first line indicates the total number of candidate sensors. An error occurs if this count mismatches the number provided in the sensor labels file. Each following line represents a candidate sensor. The first column represents the unique sensor ID (ranging from 0 to N, where N is the total sensor count), followed by n columns of transfer map indices.

Let's provide an example. Consider a structure with two possible sensor locations, each with three measurable directions (DoFs). The goal is to optimize a combination of triaxial sensors and uniaxial sensors. In this example, assume that the uniaxial sensors can only measure the x-direction. There are four candidate sensors: two triaxial sensors, and two uniaxial sensors. Let's assume the rows of the transfer matrix $\mathbf{H}$ are ordered as

$$\mathbf{H} = \begin{bmatrix} \text{Node 1, DoF x} \\ \text{Node 1, DoF y} \\ \text{Node 1, DoF z} \\ \text{Node 2, DoF x} \\ \text{Node 2, DoF y} \\ \text{Node 2, DoF z} \end{bmatrix}$$

Then in this example, the transfer map is given by

```
Example Transfer Map File
     4 // Total Candidate Sensors
     0  0  1  2 // Node 1, Dofs x,y,z
     1  3  4  5 // Node 2, Dofs x,y,z
     2  0  // Node 1, Dof x only
     3  3  // Node 2, Dof x only
```

The corresponding sensor labels would look like

```
Example Transfer Map File
     4
     0
     0
     1
     1
```

Now consider a second example where the candidate uniaxial sensors could measure any of the three directions. In this example, the transfer map file would look like

```
Example Transfer Map File
     8  // Total Candidate Sensors
     0  0  1  2 // Node 1, Dofs x,y,z
     1  3  4  5 // Node 2, Dofs x,y,z
     2  0  // Node 1, Dof x only
     3  3  // Node 2, Dof x only
     4  1  // Node 1, Dof y only
     5  4  // Node 2, Dof y only
     6  2  // Node 1, Dof z only
     7  5  // Node 2, Dof z only
```

and the corresponding sensor labels are given as

```
Example Transfer Map File
     8
     0
     0
     1
     1
     1
     1
     1
     1
```

Multi-Budget: Overlapping Nodes

By default, the greedy algorithm does not select two sensors with overlapping dofs. This implemenation prevents the algorithm from placing a uniaxial sensor and triaxial sensor at the same location. Consider the previous example. If the algorithm selects the triaxial sensor with ID 1, which measures dofs 3, 4, 5, then the uniaxial sensors with IDs 3, 5, 7 will not be selected. To turn off this behavior, set **Allow DOF Overlap** to true.

```
Input Deck: Sensor Overlap
     ParameterList name = "Greedy"
          Parameter name = "Allow DOF Overlap" value = "true"
     ParameterList
```

When **Allow DOF Overlap** is true, then the algorithm can select two sensors that share DoFs.

Multi-Budget: Budget Specification

We use the **Multiple Budgets** parameter list to specify a sensor budget for each sensor type. The budget label corresponds to the sensor label provided in the sensor label file.

```
Input Deck: Multiple Budget
     ParameterList name = "Budgets"
          Parameter name = "Sensor Type 0" int = "3"
          Parameter name = "Sensor Type 1" int = "2"
          ⋮
          Parameter name = "Sensor Type N" int = "n"
     ParameterList
```

4.6.3. Greedy Mean Squared Error Objective Functions

In addition to the classical alphabet criteria, the greedy algorithm also supports mean squared error (MSE) and mean squared prediction error (MSPE) objective functions.

> MSE and MSPE are currently enabled for the Greedy algorithm ONLY.

The MSE objective function is specified using the **OED** parameter list. MSE is computed in a Monte Carlo fashion by sampling two random variables. The first is a prior distribution on the inverse problem solution. The second is an additive measurement noise term. The additive noise is an IID Gaussian with zero mean and variance provided by the user. The objective and variance are specified in the XML file as shown below.

```
Input Deck: OED MSE
ParameterList name = "OED"
    Parameter name = "Optimality Type" value = "MSE"
    Parameter name = "Noise Variance" value = "1.0"
ParameterList
```

The solution prior is specified as a list of samples under the **Sampler** parameter list. The sample files are provided in a sampler parameter list. The user specifies the total number of samples, which must be equal to the total number of rows in the sample file. The user also specifies a weights file, which can simply be one column of ones with a total number of rows equal to the

total number of samples. This file is currently ignored, but still must have consistent dimensions. If the samples are complex then two files are provided: one for the real part and another for the imaginary part. The sampler parameter list is shown below.

```
Input Deck: MSE Sampler
ParameterList name = "Sampler"
    Parameter name = "Type" value = "User Defined Complex"
    Parameter name = "Number of Samples" value = "# of Samples"
  ParameterList name = "User Defined Complex"
    Parameter name = "Points File Real" value = "realSamples.txt"
    Parameter name = "Points File Imag" value = "imagSamples.txt"
    Parameter name = "Weights File" value = "weights.txt"
  ParameterList
ParameterList
```

Let's provide some additional details for the sample text files. The total number of rows of the sample list corresponds to the number of prior samples. Each row of the samples file corresponds to a single realization of the prior with n total columns, where n is the number of unknown parameters. In other words, n is the total number of columns of the provided transfer matrix. If the problem contains multiple observations (e.g. multiple frequencies for frequency domain type problems), then the number of columns is equal to the number of observations *times* the number of parameters. For example, consider a source inversion problem in the frequency domain with three unknown source locations and two frequencies. Let $x_{ij}^{(k)}$ represent a sample where i corresponds to the i^{th} source location, j corresponds to the j^{th} frequency, and k corresponds to the k^{th} sample. N is the total number of samples. Then the sample file takes the following form:

$$\mathbf{X} = \begin{bmatrix} x_{11}^{(1)} & x_{21}^{(1)} & x_{31}^{(1)} & x_{12}^{(1)} & x_{22}^{(1)} & x_{32}^{(1)} \\ & & & \vdots & & \\ x_{11}^{(N)} & x_{21}^{(N)} & x_{31}^{(N)} & x_{12}^{(N)} & x_{22}^{(N)} & x_{32}^{(N)} \end{bmatrix}.$$

The MSPE objective function targets prediction error. The MSPE objective is enabled in a similar fashion to the MSE objective,

```
Input Deck: OED MSPE
ParameterList name = "OED"
   Parameter name = "Optimality Type" value = "MSPE"
   Parameter name = "Noise Variance" value = "1.0"
ParameterList
```

Additionally, the user can specify specific degrees of freedom (DoFs) to compute the MSPE objective at. The specified DoFs are provided as a text file through the following parameter list interface.

```
Input Deck: MSPE Prediction Dofs
ParameterList name = "OED"
```

```
 Parameter name = "Optimality Type" value = "MSPE"
 Parameter name = "Noise Variance" value = "1.0"
 ParameterList name = "MSE"
  Parameter name = "Use Prediction Dofs" value = "true"
  Parameter name = "Prediction Dofs File" value = "predDofs.txt"
 ParameterList
ParameterList
```

In the prediction DoFs file, each listed index corresponds to a row in the transfer matrix. The index corresponding to the first row of the transfer matrix is zero. The first row of the prediction DoFs file should be the total number of prediction DoFs.

```
Example Prediction DoFs File
4 // Total number of prediction DoFs
0 // 1st row of transfer matrix
3 // 4th row of transfer matrix
10 // 11th row of transfer matrix
53 // 54th row of transfer matrix
```

Unlike the MSE objective, the MSPE objective can handle a different noise-generating model. One can provide a second transfer matrix that is used to generate data. The number of parameters in the data-generating transfer matrix does not need to equal to the number of parameters in the OED transfer matrix. To specify a data-generating transfer matrix, first indicate the Data Model is on.

```
Input Deck: MSPE Data Generating Model
ParameterList name = "OED"
   Parameter name = "Optimality Type" value = "MSPE"
   Parameter name = "Noise Variance" value = "1.0"
    ParameterList name = "MSE"
     Parameter name = "Data Model" value = "true"
    ParameterList
ParameterList
```

With Data Model set to true, the OED algorithm will look for a model labeled *Data Model* in the XML file. The *Data Model* is identical to how we specify the transfer matrix. The number of outputs and frequencies must match the OED transfer matrix, however the number of parameters may be different. When data model is on, the parameter provided in the sample list will propagate through the provided data model. Hence the number of parameters in the data model must be consistent with the provided parameter samples. An example of the *Data Model* interface is shown below.

```
Input Deck: MSPE Data Model
ParameterList name = "Data Model"
Parameter name = "Domain Type" value = "Frequency"
Parameter name = "Number of Parameters" value = "3"
```

```
Parameter name = "Number of Outputs" value = "10"
ParameterList name = "Frequency Domain"
 Parameter name = "Complex Valued" value = "true"
 Parameter name = "Number of Frequencies" value = "2"
 Parameter name = "Real Frequency Response Function" value = "realF"
 Parameter name = "Imaginary Frequency Response Function" value = "imagF"
ParameterList
ParameterList
```

In the previous two MSPE variants, the user provides a list of parameter samples where the response y is computed as ($y = H\theta$), and θ represents the parameters. However, the OED app also allows the user to specify samples of the response y directly, instead of providing parameter samples.

To enable this feature, include the **Response Data Only** option under the **MSE** parameter list as shown below:

```
Input Deck: MSPE Response Data Only
ParameterList name = "OED"
   Parameter name = "Optimality Type" value = "MSPE"
   Parameter name = "Noise Variance" value = "1.0"
    ParameterList name = "MSE"
     Parameter name = "Response Data Only" value = "true"
    ParameterList
ParameterList
```

If the **Noise Variance** value is non-zero, then that level of Gaussian noise will be added to the provided response samples. When using the **Response Data Only** option, the user provides response samples in the same format as parameter samples. Specifically, using the **User Defined Complex Sampler**, each row in the input file corresponds to a single sample of the response. For cases with multiple observations (e.g., frequency-dependent samples), all observations for a given sample are concatenated along the row. For example, if there are three samples and two frequencies, denoted as:

$$y_i(\omega_j), \quad i = 1, 2, 3 \quad j = 1, 2,$$

then the samples text file will be formatted as follows:

```
Response Only Samples File
y_1(omega_1) y_1(omega_2)
y_2(omega_1) y_2(omega_2)
y_3(omega_1) y_3(omega_2)
```

Response Data Only does not support a Transfer Map. Error will be thrown if Transfer Map provided.

4.7. Input Optimization with Greedy

In some cases, the user may wish to optimally select the columns of the transfer matrix. We call this problem input optimization or source placement. For example, in multi-axis vibration testing, designers are interested in placing electrodynamic shakers in order to excite the system of interest.

InverseOED can be used for input optimization with the E-criterion, D-criterion and MSPE response only objective functions. To enable input optimization, simply provide the **Use Input Optimization** option under the **Greedy** parameter list as shown below:

```
Input Deck: Greedy Input Optimization
ParameterList name = "Greedy"
     Parameter name = "Use Input Optimization" value = "true"
ParameterList
```

4.8. Robust Model OED

The greedy algorithm also supports multiple models. This feature can be used to optimize a single set of sensors for multiple models to achieve sensor locations that are robust to expected changes in models such as changes to boundary conditions or changes to subassemblies. There are two **Risk Type** options to select from: **Neutral** and **Averse**. The risk neutral option minimizes the average D-criteria (or E-criteria) across all models, while the risk averse option minimizes the maximum D-criteria across all models.

The user specifies the risk type and total number of models in the **Greedy** sublist as shown below

```
Input Deck: Robust Multi-Model OED
ParameterList name = "Greedy"
     Parameter name = "Risk Type" value = "Neutral" (Averse)
     Parameter name = "Linear Model Total" value = "10"
ParameterList
```

When multiple models are specified (i.e. **Linear Model Total** is greater than one), then the user is expected to provide corresponding *Linear Model* sublists for each model. These models are distinguished by appending the sublist with the corresponding number. For example, if the user lists only two models then the following sublists are needed:

```
Input Deck: Multi-Model List
ParameterList name = "Linear Model 1"
     Provide parameters for linear model one
ParameterList
ParameterList name = "Linear Model 2"
```

```
      Provide parameters for linear model two
ParameterList
```

By default, the risk neutral option is used. For frequency domain problems, each frequency is treated in a manner similar to a model. When risk neutral is used, the algorithm minimizes the D/E criterion averaged across frequencies. When risk averse is used, the algorithm minimizes the maximum D/E criterion over all frequencies. Note that this still holds true even for the deterministic setting when only one *Linear Model* is provided.

5. TRACE

5.1. Introduction

TRACE (**T**RACE **R**apidly **A**cquires **C**ontour **E**stimates) is currently in beta release and under active development. The purpose of TRACE is to efficiently estimate inputs to a black box model (aka high fidelity model) that will result in some failure criterion. For example, these inputs may be the material characteristics in a structural FEM model and the failure criterion might be exceeding some internal deformation threshold. We call the hypersurface separating these inputs from inputs that do not result in failure the 'decision boundary'. For example, this decision boundary is given by the dashed line in Figure 5-1, which separates the input combinations that result in failure (shown in red) from the input combinations that don't produce failure (shown in green).

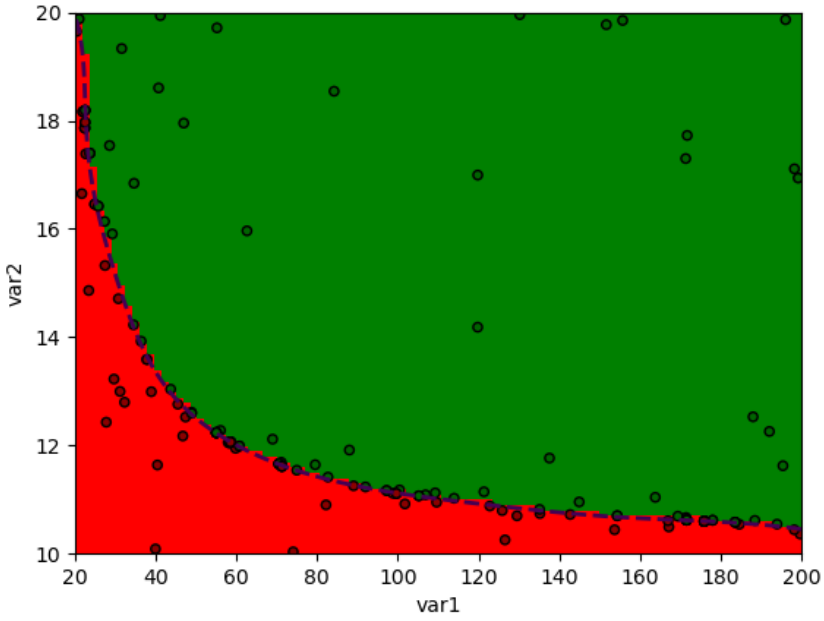


Figure 5-1. – Example decision boundary. Red indicates region of predicted failure. Each point represents a training sample used by TRACE

Currently, TRACE is based on the machine learning technique of support vector machines (SVMs). See references [1, 2] for details. TRACE efficiently estimates the decision boundary by actively querying the high fidelity model over multiple iterations, adaptively learning which input

combinations result in the user-defined failure condition and which do not. Read further to learn the many ways the TRACE algorithm can be adjusted for different use cases.

5.2. Minimal Working Example

The code below provides a minimal working example. There are two required files: the input deck `input.in` and the Python implementation (or wrapper) of the high fidelity model in `Model.py`. To run the tool, navigate to the directory where the files exist and run the following command:

```
fused_trace.py train -i input.in
```

A directory called `output` will be created where the trained classifier (`model.pickle`), a log file (`trace.log`), and other outputs will be saved. The input deck `input.in` contains all the configuration details required to run TRACE:

```
"iterations" = 100
"tolerance" = 1e-3
"background samples" = 5000
"test samples" = 5000
"initial training samples" = 10

# Required only for repeatability
"seed" = 65537

[variables]
"x" = [-2, 2, "uniform"]
"y" = [-2, 2, "uniform"]

[model]
"file" = "Model.py"
```

Input 5.2.1. input.in

The `Model.py` file contains the user-specified high fidelity model. The user's model must implement the `predict` function, which accepts a 1D array of numbers and returns either 0 (pass) or 1 (fail). If the user prefers, this function may return a continuous value, the threshold would be defined in the input deck and TRACE would handle applying the cutoff to determine the class 0/1.

```
class Model:
    def predict(self, point):
        x = point[0]
        y = point[1]
        return 1 * ((x**2 + y**2) > 1.0)
```

Input 5.2.2. Model.py

Rather than call the tool from the command line, users can also call the tool from a Python script. For example:

```
import sys
sys.path.append("path/to/src")
from fused_trace import trace

if __name__ == "__main__":
    trainer = trace("input.in")
```

By default, the results of TRACE are stored inside the `output` directory. This directory contains the training data that the surrogate used, a pickled version of the surrogate itself, and a log file that describes TRACE's progress while training. Users are able to query a set of data points for their predicted class labels using the trained surrogate. To access the surrogate model trained by TRACE, run the following command to predict the labels of the samples named `dataPoints` stored in `input_data.mat`:

```
predict.py -i input_data.mat -m output/model.pickle -n dataPoints
```

For more information on any of the command line interfaces, run `fused_trace.py -h` or `predict.py -h`.

5.3. Input Deck Format

An input deck is required to configure and run TRACE. The format of the input deck is TOML (Tom's Obvious Minimal Language). The general format for specifying a configuration parameter is

```
"option name" = int, float, bool, "str", [list], or {dictionary_key = val}
```

Strings with spaces in them *must* be enclosed by single or double quotes. It is recommended always to enclose strings in quotes to avoid inadvertent errors. Lists use the format typical of Python lists, with opening and closing brackets and with elements separated by commas. Dictionaries are similar but use braces in place of brackets. Lists and dictionaries may be nested and can span multiple lines.

5.3.1. Algorithmic Parameters

Both the required and optional algorithmic parameters *must* appear at the beginning of the input deck, before the `[variables]`, `[model]`, and `[output]` sections.

5.3.1.1. Required

The input deck has several required top-level entries or else the tool will raise an error. These are given in Table 5-1 and described in detail below.

Parameter	Type
background samples	int $\in [1, \infty)$
test samples	int $\in [1, \infty)$
initial training samples	int $\in [1, \infty)$
iterations	int $\in [0, \infty)$
tolerance	float $\in [0, 1]$

Table 5-1. – Required Algorithmic Parameters

background samples The number of background samples from which the adaptive training algorithm draws when selecting new points to train on.

test samples The number of background samples that the adaptive training algorithm uses to check for convergence.

initial training samples The number of initial training samples that the adaptive trainer attempts to train on.

iterations The number of maximum iterations TRACE will use for training. If 0, only the initial samples will be used for training.

tolerance The tolerance used for the convergence criteria. Closer to 0 means more difficult to converge. We suggest 1e-5.

5.3.1.2. Optional

In addition to the required top-level input deck parameters, there are several optional top-level parameters that the user can specify. These are given in Table 5-2 and described in detail below.

Parameter	Type	Default
adaptive algorithm	str $\in$ {"greedy", "basmis", "SAC", "greedyAC"}	"greedy"
regularization	float $\in (0, \infty)$	1000
restart	bool	false
restart filename	str	initialTrainingData.mat
plot frequency	int $\in [0, \infty)$	0
num plot steps	int $\in [1, \infty)$	100
plot samples	bool	true
run in parallel	bool	false
number of processors	int $\in [1, \infty)$	All available
iteration concurrency	int $\in [1, \infty)$	1
vertex	$N \times 2$ list of coordinates in the input space	None
kernel	str $\in$ {"linear", "poly", "rbf", "sigmoid"}	"rbf"
sampling method	str $\in$ {"random", "LHS"}	"random"
seed	int $\in [0, 4294967295]$	None
enrich samples	bool	false

Table 5-2. – Optional Algorithmic Parameters

adaptive algorithm The adaptive learning algorithm TRACE uses. The options are "greedy", "basmis", "SAC", and "greedyAC".

regularization The regularization TRACE will use for the SVM classification. The higher this number is, the more the classes will be strictly separated, but the longer each iteration will take. 1000 is a good balance for most problems.

restart If set to `true`, TRACE will read in a matlab file and set the initial training points from the matlab file. If labels are not provided, it will run the HiFi model to ascertain them.

restart filename The name of the matlab file that the **restart** option uses. The matlab variables should be

- `training` (required): $N \times M$ array, N number of points, M dimensions of input parameter space
- `labels` (optional): $N \times 1$ array containing labels for the training points. 1 means fail, 0 means pass.

plot frequency At the end of training, a convergence plot is automatically generated. For cases with exactly two input parameters, a plot of the decision boundary is also automatically generated. The user may optionally request that these plots be produced after every N^{th} iteration using this option. A value of 0 means only generate plots after training has completed.

num plot steps The resolution of the decision boundary plot generated for 2D cases. For N plot steps, an $(N+1) \times (N+1)$ grid is used for plotting. Each axis is divided into the same number of steps, regardless of the input parameter ranges.

plot samples If enabled, the samples used for training the classifier will be plotted as points on top of the colored regions in the decision boundary plot. The colored regions are always plotted for 2D cases.

run in parallel If enabled, the multiple points that the adaptive algorithm identifies as the next training points will be evaluated in parallel rather than one at a time.[1]

number of processors How many processors to use for parallel evaluation of new training points. If not specified, all available processors will be used. Unused if **run in parallel** is not activated.

iteration concurrency How many iterations into the future to look when deciding which new training points to evaluate. A value of 1 means the default sequential adaptive learning. **NOTE:** The number of points that get passed to the HiFiModel will vary depending on the number of classes and the algorithm chosen, but generally grows quite quickly. For example, using the greedy algorithm with two classes, a concurrency of 1 means 2 points must be evaluated; a concurrency of 2 means 10 points must be evaluated; a concurrency of 3 means 42 points must be evaluated; a concurrency of 4 means 170 points must be evaluated.

vertex If supplied, only the polygonal subspace defined by the vertices will be considered when selecting new training points to evaluate. **NOTE:** Only supported in 2D and vertices *must* be in the correct order.

kernel The kernel used for kernelized SVM. If set to `"linear"`, the classifier is equivalent to a standard SVM. We *strongly* recommend the default of `"rbf"`.

sampling method Method used for generating random samples. `"random"` (the default) will use simple uniform random sampling of the ranges specified in the `[variables]` block (see below). `"LHS"` instead uses a custom implementation of Latin hypercube sampling.

seed The seed used for random number generation within TRACE. Specifying a value will ensure repeatable results. If unset, a random seed is chosen.

enrich samples Boolean that specifies whether to add additional samples near the boundary for the purposes of visualization. If true, once the surrogate is trained, additional samples will be placed near the decision boundary, whose labels are predicted by the classifier. These points will be saved in the output directory under "enrichedSamples.mat".

[1] **IMPORTANT**: The current implementation of parallelism does not guarantee that a model's internal state is preserved. For example, if your model uses an internal counter (e.g., `self.iter += 1`), this counter may be reset arbitrarily. It is therefore recommended to write your model in such a way that it avoids using any internal state for `predict()`.

5.3.2. Variables

Specifying the variables is required, although it's done separately from the other options. Under the `[variables]` header of the input deck, each line will have a variable, e.g. `var1 = [3, 7, "uniform"]` creates the variable `var1` $\sim$ Uniform$[3, 7]$. The variables have names, and are ordered from top to bottom when reading in data. That is, order *does* matter and the user's high-fidelity model should expect the variables in the order in which they occur in the input deck. Each variable consists of a list of three elements: the first distribution parameter, the second distribution parameter, and the name of the distribution. A list of supported distributions and their corresponding variables are in Table 5-3.

Distribution	Variable List
`var` $\sim$ Uniform$[a, b]$	`var = [`a, b`, "uniform"]`
`var` $\sim$ Normal(μ, σ)	`var = [`μ, σ`, "normal"]`
`var` $\sim$ Beta(α, β)	`var = [`α, β`, "beta"]`

Table 5-3. – Supported Variable Distributions

5.3.3. Model Parameters

It is required to specify the high-fidelity model that will be used by TRACE. Model specification is done under the `[model]` header. The full list of model parameters is in Table 5-4.

Parameter	Type	Required
file	str	Yes
type	str	No
class	str	No
parameters	dict	No
threshold	float	No

Table 5-4. – Model Parameters

file The file path (relative or absolute) to the model to load.

type Type of high-fidelity model with which TRACE will interface. Right now, only `"python"` is supported. In the future, we hope to support models defined directly in MATLAB or a shell script.

class The name of the class to import from `file`. If omitted, defaults to the name of the file.

parameters A dictionary of parameters passed to the model's initializer. If omitted, defaults to an empty dictionary. The parameter dictionary is always passed to the model's initializer (even if it's not specified) so the model's initializer should always except an input parameters dictionary even if it's not used.

threshold A threshold value that will be applied to the output from the `predict` function in the `Model.py` file to determine the pass/fail criteria, $1 * (value > threshold)$. The default value is 0, assuming a user has provided 0/1 label values from the `predict` function.

5.3.4. Output Parameters

While output parameters are not required, the `[output]` block will allow the user to specify output file names.

Parameter	Type	Default
path	str	"output"
saved data filename	str	"trainingData.mat"
saved model filename	str	"model.pickle"
overwrite	bool	false
log_file	bool	"trace.log"
console	bool	true
verbose	bool	false

Table 5-5. – Output Parameters

path Directory for TRACE's output. An error will be raised if the directory already exists.

saved data filename Filename for the training data that will be saved.

saved model filename Filename for the surrogate model that will be saved.

overwrite Whether or not TRACE will overwrite any data contained in the specified output folder.

log_file Filename for the log file containing informational messages.

console Whether or not to print informational messages to the console (i.e., stdout).

verbose Whether or not to print extra information to the console and log file.

5.3.5. Metrics

The metrics block is optional and when enabled, TRACE will calculate the desired metrics listed. The options are "accuracy", "balanced accuracy", "precision", "recall", "specificity", and "f1". For an overview of what these metrics calculate, please check SciKit-Learn's documentation.

Parameter	Type	Default
training metrics filename	str	"trainingMetrics.csv"
training metrics	list[str]	None
validation metrics filename	str	"validationMetrics.csv"
validation metrics	list[str]	None
validation samples	str	None
validation labels	str	None
number of validation samples	int $\in [1, \infty)$	None

Table 5-6. – Metrics Parameters

training metrics filename Specify the filename for the training metrics.

training metrics Specify the training metrics you would like to save as a list of strings. The options are "accuracy", "balanced accuracy", "precision", "recall", and "f1".

validation metrics filename Specify the filename for the validation metrics.

validation metrics Same as training metrics.

validation samples Filename for the validation samples to be used, with data saved as `"X"`. If set to "random", TRACE will automatically generate samples to be used.

validation labels Filename for the validation labels to be used, with data saved as `"y"`. If **validation samples** is set to "random", this must be left empty.

number of validation samples Number of samples to be generated when **validation samples** is set to "random".

5.4. Postprocessing Probabilities

As of Sierra 5.26, this capability is not fully supported and may not work as documented. If you wish to use this capability, it is strongly recommended to use the Sierra version of the day (i.e., `sierra/daily`) to get the latest fixes. If you do not have access to the version of the day, please reach out to the FuSED team for assistance.

It may be that some of the variables in the user's high-fidelity model represent uncertain parameters. Alternatively (or additionally), the user's high-fidelity model may not output a deterministic pass/fail condition, but rather a *probability* of failure. These use cases are summarized in Figure 5-2, where we consider the model $\mu = f(\boldsymbol{\xi})$. $\boldsymbol{\xi}$ is the model input, and all, some, or none of it may be stochastic. μ is the model output, and either $\mu \in \{0, 1\}$ (for deterministic labels) or $\mu \in [0, 1]$ (for probabilities).

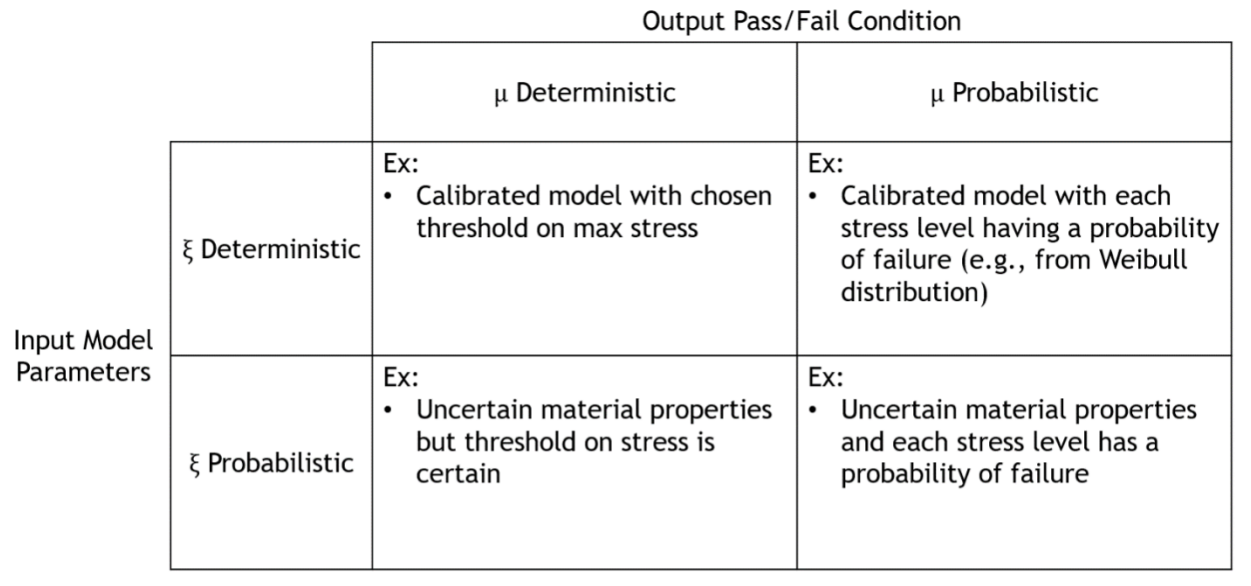


Figure 5-2. – Quad-chart of different flavors of stochastic/deterministic decision boundaries.

Rather than a classifier that gives a completely deterministic prediction (0 or 1, pass or fail), the trained surrogate model in TRACE can be postprocessed to predict probability of failure. After training on the full variable space, the user can add a `[postprocessing]` block to the input deck with the required parameters listed in Table 5-7. To run the postprocessing step, execute TRACE using the `postprocess` argument as shown below:

```
fused_trace.py postprocess -i input.in
```

A new directory named `postprocess` will be created, with a subdirectory therein for each value specified in `contours`.

5.4.1. Required Parameters

Parameter	Type
model file	str
uncertain variables	list[str]
contours	list[float] $\in (0, 1)$
number of samples	int

Table 5-7. – Required Postprocessing Parameters

model file The path to the surrogate model trained by TRACE, e.g., `output/model.pickle`.

uncertain variables Names of the variables that are considered uncertain. These names must also occur in the `[variables]` block.

contours Probabilities for which TRACE will estimate contours. For example, `[0.80, 0.90, 0.95]` specifies that TRACE should separately find the 80% failure contour, 90% failure contour, and 95% failure contour.

number of samples Number of samples used in the Monte Carlo estimation of probabilities. For details of the approach, see [1].

5.4.2. Optional Parameters

Parameter	Type	Default
iterations	int	100
enrich samples	bool	false

Table 5-8. – Optional Postprocessing Parameters

iterations Specify the number of iterations used to train the postprocessing model. Since this is done on the trained surrogate model, they are not computationally expensive.

enrich samples Boolean that specifies whether to add additional samples near the boundary for the purposes of visualization. If true, once each probability of failure contour is trained, additional samples will be placed near the decision boundary, whose labels are predicted by the classifier. These points will be saved in their respective contour folder under "enrichedSamples.mat".

5.5. Reinforcement Learning

This is a beta capability within TRACE. It requires a separate Python environment where PyTorch is installed.

There are two options available to use reinforcement learning in TRACE. Additional details about the theory behind these algorithms can be found in Netter et al. 2025 [17]. To select either the "SAC" or "greedyAC" algorithms, see the "adaptive algorithm" setting in Table 5-2.

In order to use the reinforcement learning algorithms, the `PyTorch` library must be available. This is not distributed with Sierra, so the library must either be installed locally or in an activated environment. Steps to create a custom conda environment and install the library are below.

```
conda create -n rl python=3.12
conda activate rl
pip install pytorch
```

In reinforcement learning, there is a parameter called `certainty` that controls the fidelity of simulation you wish to run. The idea is that low-quality simulations are run first to get a vague idea of the decision boundary, informing the general location where the algorithm needs to focus in and run those higher-quality simulations. In order to control the fidelity of simulation being run, the `Model.py` must be updated:

```
class Model:
    """An example of changes to the high fidelity model
    for reinforcement learning"""

    ...

    def predict(self, point, certainty = 1.0):
        """The predict function for reinfocement learning takes in
        the certainty. A high certainty indicates a higher fidelity
        model should be selected. This function cannot return a binary
        value, it should return a continuous value."""
        cutoff1 = 1.5 # The certainty will be in the range (0.2,5)
        cutoff2 = 3.5
        if certainty < cutoff1:
            f = model1(point)
        elif certainty >= cutoff1 and certainty < cutoff2:
            f = model2(point)
        elif certainty >= cutoff2:
            f = model3(point)
        return f

    ...
```

Additional settings specific to these algorithms are specified in a `[reinforcement learning]` block in the input deck, with the parameters described in Table 5-9.

5.5.1. Optional Parameters

Parameter	Type	Default
actor learning rate	float	1e-2
critic learning rate	float	5e-2
entropy weight	float	1.8
weight constant	float	1
max grad norm	float	2.5
actor sd	float	1e-1

Table 5-9. – Reinforcement Learning Parameters

actor learning rate A larger actor learning rate means that the model will learn the optimal control faster, but if the value is too large it will learn an incorrect control or overshoot and never converge.

critic learning rate A larger critic learning rate means that the model will learn the cost function faster, but if the value is too large it will learn an incorrect control or overshoot and never converge. You generally want this value to be larger than the actor learning rate.

entropy weight A higher entropy weight puts more emphasis on finding a more stochastic or "random" optimal policy, and explores more points it thinks are suboptimal. A lower entropy weight will put more emphasis on just finding an optimal point and staying there.

weight constant The weight constant affects the evaluation of the certainty where

$$certainty = max(0.2, min(5.0, 1/\text{estimated\_reward}(point)))$$

Thus, a larger weight constant indicates more certainty in the simulation indicating it is worthwhile to run a longer, higher-fidelity simulation.

max grad norm This is the upper bound for the gradient of the actor and critic. It ensures that the actor and critic do not take too large of steps when training. Therefore, this value is likely closely related to the learning rate.

actor sd This value is the initial standard deviation applied to the actor. The actor is producing a random distribution in order to select samples. Since the sample values are normalized, this value should always be less than 1.

This page intentionally left blank.

6. APPENDIX

6.1. Optimal Experiment Design Theory

6.1.1. Inverse problem framework

This section introduces an abstract framework for OED applied to linear inverse problems. To begin, we consider an abstract linear model of a system's response for the i^{th} experiment

$$y_i = h_i^\top \theta, \tag{6.1.1}$$

where $\theta \in \mathbb{K}^{n_p}$ are the unknown model parameters, $h_i^\top \in \mathbb{K}^{n_o \times n_p}$ is the parameter-to-response map associated with the i^{th} experiment, and $\mathbb{K} = \mathbb{R}$ or $\mathbb{C}$. In modal expansion and in MIMO control, the i^{th} experiment corresponds to placing a sensor at the location of the i^{th} degree of freedom (DoF), i.e., mesh vertex. In general, $h_i^\top$ is a matrix that maps the parameters θ to a set of n_o observations. For the MIMO control example, $h_i^\top$ is a block diagonal matrix that models the frequency response function and θ represents the forces acting on the system. For modal expansion, $h_i^\top$ models the system's mode shapes and θ represents the expansion coefficients. In this report, we use the standard transpose symbol $x^\top$ to refer to the complex conjugate when $\mathbb{K} = \mathbb{C}$.

Assume that we perform the i^{th} experiment $q_i \in \mathbb{N}$ times and assume that the measured response of the j^{th} instance of the i^{th} experiment satisfies the additive noise relationship

$$\tilde{y}_{i,j} = h_i^\top \theta + \epsilon_{i,j} \quad \text{for} \quad j = 1, \dots, q_i \quad \text{and} \quad i = 1, \dots, n, \tag{6.1.2}$$

where $\epsilon_{i,j} \in \mathbb{K}^{n_o}$ is a random vector representing the measurement noise and other modeling errors. In addition, we assume that the noise is homoscedastic (i.e., $\epsilon_{i,j}$ does not depend on the experiment i or the instance j) and that it is independent and identically distributed (iid) with mean zero and covariance $\sigma^2 I_{n_o}$. Here, I_k denotes the $k \times k$ identity matrix. In particular, the iid assumption ensures that

$$\mathbb{E}[\epsilon_{i,j} \epsilon_{i',j'}^\top] = \begin{cases} \sigma^2 I_{n_o} & \text{if } i = i' \text{ and } j = j' \\ 0_{n_o} & \text{otherwise} \end{cases}. \tag{6.1.3}$$

Here, 0_k denotes the $k \times k$ matrix of zeros.

Inverse problems seek to estimate θ from selected experiments. In InverseOED, we formulate inverse problems as the least-squares problem

$$\hat{\theta} \in \operatorname*{argmin}_{\theta \in \mathbb{K}^{n_p}} \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{q_i} \left\| \tilde{y}_{i,j} - h_i^\top \theta \right\|_2^2 \tag{6.1.4}$$

or

$$\hat{\theta} \in \operatorname*{argmin}_{\theta \in \mathbb{K}^{np}} \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{q_i} \left\| \tilde{y}_{i,j} - h_i^\top \theta \right\|_2^2 + R(\theta) \tag{6.1.5}$$

where $R(\theta)$ is a regularization term.

To simplify notation, we define the quantities

$$M(q) := \sum_{i=1}^{n} q_i h_i h_i^\top, \qquad Y(q) := \sum_{i=1}^{n} h_i \sum_{j=1}^{q_i} \tilde{y}_{i,j}, \qquad \text{and} \qquad E(q) := \sum_{i=1}^{n} h_i \sum_{j=1}^{q_i} \epsilon_{i,j}.$$

With this notation, the optimal solutions to the least-squares problem (6.1.4) solve the linear system of equations

$$M(q)\hat{\theta} = Y(q).$$

Nonsingular $M(q)$

If $M(q)$ is invertible, then the estimator $\hat{\theta}$ that solves (6.1.4) is unique and is given by

$$\hat{\theta} = M(q)^{-1} Y(q). \tag{6.1.6}$$

Using the model (6.1.2) and the assumptions on the noise, we further see that the estimator satisfies

$$\hat{\theta} = M(q)^{-1} \left(\sum_{i=1}^{n} q_i h_i h_i^\top \theta + E(q) \right) = \theta + M(q)^{-1} E(q),$$

where the expected value of $E(q)$ is zero since $\mathbb{E}[\epsilon_{i,j}] = 0$ for all $j = 1, \ldots, q_i$ and $i = 1, \ldots, n$. From this, we see that the expected value of the estimator $\hat{\theta}$ (averaged over the measurement noise) is $\mathbb{E}[\hat{\theta}] = \theta$. That is, $\hat{\theta}$ is an unbiased estimator of θ. Moreover, the covariance matrix of the estimated parameters in (6.1.6) is given by

$$C(q) = \mathbb{E}[(\hat{\theta} - \theta)(\hat{\theta} - \theta)^\top] = M(q)^{-1} \mathbb{E}[E(q)E(q)^\top] M(q)^{-1}.$$

From this, we notice that

$$\mathbb{E}[E(q)E(q)^\top] = \sum_{i=1}^{n} \sum_{i'=1}^{n} \sum_{j=1}^{q_i} \sum_{j'=1}^{q_{i'}} h_i \mathbb{E}[\epsilon_{i,j} \epsilon_{i',j'}^\top] h_{i'}^\top = \sigma^2 \sum_{i=1}^{n} q_i h_i h_i^\top,$$

where the final equality follows from the assumption that $\epsilon_{i,j}$ are iid with covariance matrix $\sigma^2 I_{n_o}$, cf. (6.1.3). Hence, the covariance matrix associated with the estimator $\hat{\theta}$ is given by

$$C(q) = \mathbb{E}[(\hat{\theta} - \theta)(\hat{\theta} - \theta)^\top] = \sigma^2 M(q)^{-1}. \tag{6.1.7}$$

Measures of estimation and prediction variance

The mean-squared error committed when solving the estimation problems (6.1.4) or (6.1.5) is

$$\begin{aligned}\mathbb{E}[\|\hat{\theta}-\theta\|^2] &= \mathbb{E}[\|\hat{\theta}-\mathbb{E}[\hat{\theta}]\|^2] + 2\mathbb{E}[\hat{\theta}-\mathbb{E}[\hat{\theta}]]^\top(\mathbb{E}[\hat{\theta}]-\theta) + \|\mathbb{E}[\hat{\theta}]-\theta\|^2 \\ &= \mathbb{E}[\|\hat{\theta}-\mathbb{E}[\hat{\theta}]\|^2] + \|\mathbb{E}[\hat{\theta}]-\theta\|^2, \end{aligned} \tag{6.1.8}$$

where the first term in (6.1.8) is the variance of the estimator $\hat{\theta}$ and the second is its bias. If $\hat{\theta}$ is the solution to (6.1.4), then $\mathbb{E}[\hat{\theta}] = \theta$ and the bias term is zero. On the other hand, if $\hat{\theta}$ is the solution to (6.1.5), then the bias term is given by

$$\|\mathbb{E}[\hat{\theta}]-\theta\|^2 = \left\|((M(q)+R)^{-1}M(q) - I_{n_p})\theta\right\|^2 = \left\|(M(q)+R)^{-1}R\theta\right\|^2$$

and can be bounded above by

$$\|\mathbb{E}[\hat{\theta}]-\theta\|^2 \leq \left\|(M(q)+R)^{-1}R\right\|^2 \|\theta\|^2 .$$

The first term in (6.1.8), the estimation variance, is the trace of the covariance matrix, i.e.,

$$\mathbb{E}[\|\hat{\theta}-\mathbb{E}[\hat{\theta}]\|^2] = \operatorname{tr}(C(q)),$$

and is typically referred to as the A-optimality criterion. According to the previous discussion, a reasonable approach to designing experiments is to minimize the function

$$q \mapsto \alpha \operatorname{tr}(C(q)) + (1-\alpha)\left\|(M(q)+R)^{-1}R\right\|^2$$

for a fixed convex combination parameter $\alpha \in (0,1)$. Note that if $\alpha = (1+\|\theta\|^2)^{-1}$, then this function is an upper bound for the mean-squared error, scaled by $(1+\|\theta\|^2)^{-1}$. Unfortunately, θ is typically not known, leading one to select α based on their preference to emphasize the variance or bias terms.

Using the estimator $\hat{\theta}$, we arrive at an estimator for the predicted response given by $g_i^\top\hat{\theta}$, where $g_i = h_i u$ for a user-provided vector $u \in \mathbb{K}^{n_o}$. As we did for the estimation error, we can compute the mean-squared prediction error for the i^{th} experiment as

$$\begin{aligned}\mathbb{E}[|g_i^\top\hat{\theta} - g_i^\top\theta|^2] &= \mathbb{E}[|g_i^\top\hat{\theta} - \mathbb{E}[g_i^\top\hat{\theta}]|^2] + |\mathbb{E}[g_i^\top\hat{\theta}] - g_i^\top\theta|^2 \\ &= \mathbb{E}[|g_i^\top(\hat{\theta}-\mathbb{E}[\hat{\theta}])|^2] + |g_i^\top(\mathbb{E}[\hat{\theta}]-\theta)|^2\end{aligned}$$

The first term is the prediction variance, which is given by

$$\mathbb{E}[|g_i^\top(\hat{\theta}-\mathbb{E}[\hat{\theta}])|^2] = g_i^\top C(q) g_i, \tag{6.1.9}$$

whereas the second term is the prediction bias, which is zero if $\hat{\theta}$ solves (6.1.4). When $\hat{\theta}$ solves (6.1.5), the prediction bias is given by

$$|g_i^\top(\mathbb{E}[\hat{\theta}]-\theta)|^2 = |g_i^\top((M(q)+R)^{-1}R\theta|^2$$

and is bounded above by

$$|g_i^\top(\mathbb{E}[\hat{\theta}] - \theta)|^2 \leq \|R(M(q)+R)^{-1}g_i\|^2\|\theta\|^2 = \left[g_i^\top(M(q)+R)^{-1}R^2(M(q)+R)^{-1}g_i\right]\|\theta\|^2.$$

Given a weight vector $w \in \mathbb{R}^n$ with nonnegative entries, it is reasonable to choose the vector q to minimize the function

$$q \mapsto \sum_{i=1}^{n} w_i \left(\alpha\ g_i^\top C(q) g_i + (1-\alpha)\|R(M(q)+R)^{-1}g_i\|^2\right)$$

for a fixed convex combination parameter $\alpha \in (0,1)$. The first term is called the I-optimality criterion, which quantifies the average prediction variance.

In the forthcoming section, we discuss additional optimality criteria based on both the estimation and prediction variance. In the current version, InverseOED does not support the nonsingular covariance case, which includes the regularization term and the convex combination parameter. However, these features maybe implemented for future versions of the InverseOED library.

6.1.2. Gradient-based optimization formulation

InverseOED implements two classes of algorithms for performing OED: gradient-based optimization algorithms and greedy algorithms. Each algorithm class determines the design vector q by approximately minimizing a functional of the estimated parameter covariance matrix.

We first present the *exact* design formulation of OED, which seeks a vector of nonnegative integers $q \in \mathbb{N}^n$. The entries of q correspond to the number of times each experiment is performed. Given an experiment budget $b \in \mathbb{N}$, we seek to compute an optimal design q^* by solving the integer optimization problem

$$q^* \in \operatorname*{argmin}_{q \in \mathbb{N}^n} \Psi(C(q)) \quad \text{subject to} \quad \sum_{i=1}^{n} q_i = b. \tag{6.1.10}$$

Again, the optimality criterion Ψ is a scalar function acting on matrices that quantifies estimation or prediction uncertainty. The exact design problem (6.1.10) is difficult to solve since q is a required to be a vector of nonnegative integers. To circumvent this challenge, we can set $p = q/b$ and note that if q satisfies the budget constraint in (6.1.10), then $0 \leq p_i \leq 1$ for $i = 1, \ldots, n$. If Ψ is positively homogeneous (a condition that is typically satisfied by optimality criteria), then substituting $q = bp$ with $p \in [0,1]^n$ into (6.1.10) produces the *approximate* design problem

$$p^* \in \operatorname*{argmin}_{p \in [0,1]^n} \Psi(C(p)) \quad \text{subject to} \quad \sum_{i=1}^{n} p_i = 1. \tag{6.1.11}$$

The components of an optimal solution p^* to (6.1.11) represent the frequency for which each experiment is run. Multiplying p^* by the budget b then produces an approximate design, which can be rounded to the nearest integer value to produce a schedule of experiments.

For many common optimality criteria Ψ, the objective function $\Psi(C(p))$ in (6.1.11) is differentiable and convex. In this case, the globally optimal solution to (6.1.11) can be computed using a gradient-based optimization. The InverseOED app uses the Rapid Optimization Library to formulate and solve (6.1.11) [20]. Note that from a practical point of view, the experiment schedule q are only realizable if it is feasible to perform an experiment multiple times and we note that running an experiment multiple times can reduce uncertainty in the estimated parameters by improving the estimate of the noise in that experiment.

6.1.3. Greedy-based optimization formulation

For the greedy formulation, we consider only binary designs, i.e., $q_i \in \{0, 1\}$ for $i = 1, \ldots, n$. This models the situation in which each experiment can be performed only once. The greedy algorithm, Algorithm 1, approximately solves the binary optimization problem

$$q^* \in \underset{q\in\{0,1\}^n}{\operatorname{argmin}} \ \Psi(C(q)) \quad \text{subject to} \quad \sum_{i=1}^{n} q_i = b. \tag{6.1.12}$$

In the desciption of Algorithm 1, we employ the notation $\widehat{\Psi}(S)$, where S is a subset of integers between 1 and n, to denote $\Psi(C(q))$, where $q_i = 1$ if $i \in S$ and $q_i = 0$ otherwise.

Algorithm 1 The Greedy Algorithm

1: **Initialization**: $S = \emptyset$ and $M = 0$
2: **while** $M < b$ **do**
3: $\quad j^* = \underset{j\in\{1,\ldots,n\}\setminus S}{\arg\min} \ \widehat{\Psi}(S \cup \{j\})$
4: $\quad S = S \cup \{j^*\}, \quad M = M + 1$
5: **end while**

Although the greedy approach is a heuristic and results in a suboptimal solution, it is known to work very well. In cases where the objective function is submodular it can be proven that the greedy solution produces an objective function value that is within $(1 - e^{-1})$ of the optimal objective value, where e is Euler's constant [19]. Currently, the greedy algorithm implemented in InverseOED is specialized for D-optimalty; one reason being that the D-optimality criterion is submodular. We note that, in contrast to (6.1.11), the greedy algorithm naturally enforces the sensor budget simply by terminating the search once the budget is reached.

6.1.4. Optimality criteria

The goals of an inverse-based experiment are to use measured data to either *(i) estimate* the parameters of a model or *(ii) predict* the unobserved response using the estimated parameter. By choosing the optimality criterion Ψ in the objective function of (6.1.11), we can achieve experiment designs that minimize either parameter or prediction uncertainty. Table 6-1 provides a

Table 6-1. – Optimality Criteria

Criterion	$\Psi(C)$	*Description*
A	$\mathrm{Tr}(C)$	Average estimation variance
C	$v^\top C v$	Variance of $v \in \mathbb{R}^m$ times the estimator
D	$\det(C)$	volume of the covariance
I	$\mathbb{E}[g^\top C g]$	Average prediction variance
R	$\mathrm{AVaR}_\beta[g^\top C g]$	Tail average prediction variance for $\beta \in [0,\ 1]$

list of the optimality criteria that are implemented in the gradient-based version of the InverseOED app.

Based on our definition of experiment, we understand the prediction variance to mean the variance of $u^\top h_i^\top \hat{\theta} = g_i^\top \hat{\theta}$, where $u \in \mathbb{K}^{n_o}$ is a user-defined vector (e.g., the vector of ones). It then follows that the prediction variance is given by (6.1.9). We treat this quantity as a random variable with respect to the sensor locations (DoFs). This interpretation allows us to formulate the I- and R-optimality criteria in order to minimize statistics of the prediction uncertainty.

In the context of unregularized least-squares (6.1.4), it is customary to define Ψ so that $\Psi(C(q)) = +\infty$ if $M(q)$ is singular. For example, the D-optimality criterion is given by $\Psi(C) = \det(C)$. Using the form of $C(q)$ when $M(q)$ is invertible, we can rewrite $\Psi(C(q))$ as

$$\Psi(C(q)) = \sigma^{2n_p} \det(M(q))^{-1}.$$

In this form, $\Psi(C(q))$ is defined even when $M(q)$ is singular; in which case, $\Psi(C(q))$ is infinite.

A-optimality

The A-optimality criterion seeks to minimize the estimation variance of $\hat{\theta}$ and is given by

$$\Psi(C) = \mathrm{tr}(C) = \sum_{i=1}^{n_p} C_{ii} \tag{6.1.13}$$

C-optimality

The C-optimality criterion seeks to minimize the variance of $v^\top \hat{\theta}$ for user-provided $v \in \mathbb{K}^{n_p}$ and is given by

$$\Psi(C) = v^\top C v. \tag{6.1.14}$$

D-optimality

The D-optimality criterion seeks to minimize the volume of the parameter uncertainty ellipsoid associated with $\hat{\theta}$ and is given by

$$\Psi(C) = \det(C) \tag{6.1.15}$$

In the case of homoscedastic noise, D- and G-optimality are equivalent [11], where G-optimality minimizes the maximum prediction variance over all experiments.

I-optimality

The I-optimality criterion seeks to minimize the average prediction variance over all candidate experiments. Given a nominal probability distribution defined on the set of experiments $\{1, \ldots, n\}$ with probabilities $w_k \geq 0$, the I-optimality criterion is given by

$$\Psi(C) = \sum_{k=1}^{n} w_k g_k^\top C g_k \tag{6.1.16}$$

For example, setting $w_k = 1/n$ for $k = 1, \ldots, n$ produces the I-optimality criterion associated with the uniform distribution of experiments. Owing to the linearity and cyclic invariance properties of the trace, we can rewrite the I-optimality criterion as

$$\Psi(C) = \sum_{k=1}^{n} w_k \mathrm{tr}\left(g_k^\top C g_k\right) = \mathrm{tr}\left(C \underbrace{\left[\sum_{k=1}^{n} w_k g_k g_k^\top\right]}_{=:B}\right) = \mathrm{tr}\left(CB\right).$$

By computing the matrix B offline, the online computational expense for solving the I-optimal design problem is comparable to solving the A-optimal design problem.

R-optimality

In general, G-optimality can be overly conservative as it minimizes the worst-case prediction variance, while I-optimality does not penalize heavy tailed statistics. R-optimality is a new risk-adapted optimality criterion that seeks a design that represents a trade-off between G-optimality and I-optimality [13, 14]. R-criteria is the average value-at-risk of the prediction variance for a provided confidence level $\beta \in (0, 1)$. Given a nominal distribution of experiments with probabilities $w_k \geq 0$, the R-optimality criterion is defined as

$$\Psi(C) = \min_{t \in \mathbb{R}} \left\{ t + \frac{1}{1-\beta} \sum_{k=1}^{n} w_k \max\{0, g_k^\top C g_k - t\} \right\} \tag{6.1.17}$$

The average value-at-risk is a statistical measure of the tail of a random variable's distribution. It can be thought of as the the average of the $(1-\beta)\times 100\%$ largest scenarios for a fixed confidence level $\beta \in (0,1)$. For clarity, Figure 6-1 presents the probability distribution function of an example prediction variance distribution and highlights the R-optimality value, which equals the average taken over the shaded region. The user's aversion to risk is reflected in the specified confidence level, which sets the quantile at which the average value-at-risk is computed. It is instructive to understand the cases when $\beta = 0$ and $\beta = 1$. As β approaches one, the R-optimality criterion approaches the G-optimality criterion whereas if $\beta = 0$, the R-optimality criterion is the I-optimality criterion.

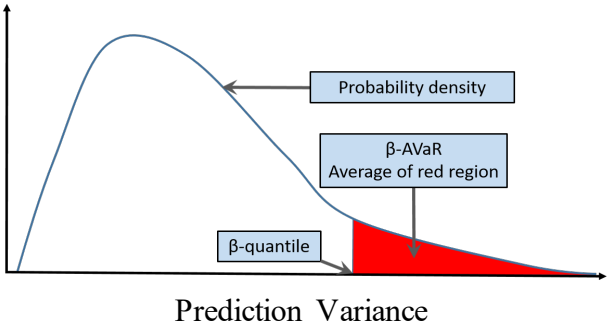


Figure 6-1. – An example probability distribution function of the prediction variance where the R-criteria equals the average taken over the shaded region

6.1.5. Structural dynamics inverse problem examples

We apply InverseOED to two structural dynamics inverse problems: modal expansion and MIMO source inversion. This section provides a brief overview of these problems. For additional details see [16].

Modal expansion

We begin by modeling the structure's dynamic response as the solution to the system of differential equations defined by the following governing equations of motion

$$\mathbf{M}\ddot{u}(t) + \mathbf{C}\dot{u}(t) + \mathbf{K}u(t) = f(t) \tag{6.1.18}$$

where $\mathbf{M} \in \mathbb{R}^{NxN}$, $\mathbf{C} \in \mathbb{R}^{NxN}$, and $\mathbf{K} \in \mathbb{R}^{NxN}$ are respectively the mass, damping, and stiffness matrices. In cases where the structure is lightly damped, the mode shapes $\phi_r \in \mathbb{R}^N$ for $r = 1, 2, \ldots, N$ can be found from the solution of the eigenvalue problem

$$(\mathbf{M}^{-1}\mathbf{K} - \lambda_r \mathbf{I})\phi_r = 0 \tag{6.1.19}$$

The mode shapes form a basis for the dynamic response $u(t_k)$ at any instance of time t_k. To simply notation, let $u := u(t_k)$. We assume that u can be approximated by the span of a subset of

n_α mode shapes, where $n_\alpha < N$. Then by definition, there exists a set of expansion coefficients $z \in \mathbb{R}^{n_\alpha}$ such that

$$u_i \approx \tilde{u}_i = \sum_{j=1}^{n_\alpha} \phi_{i,j} z_j \tag{6.1.20}$$

The modal expansion inverse problem seeks to estimate $z := z(t_k)$ from the measured data. We apply the inverse and associated OED framework to the modal expansion problem by transforming (6.1.20) into the form of (6.1.1) using the following substitutions,

$$y_i := \tilde{u}_i, \qquad h_i^\top := \phi_i^\top, \qquad \text{and} \qquad \theta = z$$

Note that the covariance of the estimated expansion coefficients is time independent since the mode shapes are independent of time .

MIMO control

We now turn to our application of the OED to MIMO control problem in the temporal frequency domain. The control problem entails estimating loads at a subset of $n_f \leq N$ DoFs associated with the control input locations in order to generate the measured response. We denote a particular frequency in the temporal frequency domain by $\omega \in \mathbb{R}$. The frequency domain transformation of (6.1.18) is given by the following system of algebraic linear equations

$$\hat{u}(\omega) = A(\omega)\hat{f}(\omega). \tag{6.1.21}$$

where $A(\omega) \in \mathbb{C}^{N\times N}$ is the frequency response function (FRF) defined as

$$A(\omega) = (\mathbf{K} - j\omega\mathbf{C} + \omega^2\mathbf{M})^{-1}. \tag{6.1.22}$$

Let $\bar{f}(\omega) \in \mathbb{C}^{n_f}$ denote the forces associated with the control input DoFs and $a_i^\top(\omega) \in \mathbb{C}^{n_f}$ represent the appropriately indexed FRF that maps the control forces to the response $\hat{u}_i(\omega) \in \mathbb{C}$. It follows that the response at DoF i is given by

$$\hat{u}_i(\omega) = a_i^\top(\omega)\bar{f}(\omega) \tag{6.1.23}$$

Assuming the frequency domain is discretized into n_ω frequencies, we can express the MIMO control OED problem in a form compatible with (6.1.1) by making the following substitutions

$$\begin{aligned} y_i &:= [\hat{u}_i(\omega_1), \hat{u}_i(\omega_2), \ \ldots, \hat{u}_i(\omega_n)] \\ h_i^\top &:= \begin{bmatrix} a_i^\top(\omega_1) & 0 & \cdots & 0 \\ 0 & a_i^\top(\omega_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_i^\top(\omega_{n_\omega}) \end{bmatrix} \\ \theta &:= [\bar{f}(\omega_1), \bar{f}(\omega_2), \ \ldots, \bar{f}(\omega_{n_\omega})]. \end{aligned} \tag{6.1.24}$$

Note in the MIMO control problem, the number of observations n_o is equal to the number of frequencies n_ω.

This page intentionally left blank.

BIBLIOGRAPHY

[1] W. Aquino et al. “Assessing Decision Boundaries Under Uncertainty”. In: *Structural and Multidisciplinary Optimization* 67.7 (2024) (cit. on pp. 101, 111).

[2] Wilkins Aquino et al. *Support Vector Machines for Estimating Decision Boundaries with Numerical Simulations*. Technical report SAND2022-11061. Albuquerque, New Mexico 87185: Sandia National Laboratories, Aug. 2022 (cit. on p. 101).

[3] Martin P Bendsøe. “Optimal shape design as a material distribution problem”. In: *Structural optimization* 1.4 (1989), pp. 193–202 (cit. on p. 53).

[4] D. P. Bertsekas. “Projected Newton Methods for Optimization Problems with Simple Constraints”. In: *SIAM J. Control and Optimization* 20 (1982), pp. 221–246 (cit. on p. 31).

[5] Gregory Bunting et al. “Novel Strategies for Modal-Based Structural Material Identification”. In: *Journal of Mechanical Systems and Signal Processing* 149.107295 (2021) (cit. on pp. 11, 12).

[6] J. V. Burke, J. J. Moré, and G. Toraldo. “Convergence properties of trust region methods for linear and convex constraints”. In: *Math. Programming* 47 (1990), pp. 305–336 (cit. on p. 31).

[7] P. H. Calamai and J. J. Moré. “Projected gradient methods for linearly constrained problems”. In: *Math. Programming* 39 (1987), pp. 93–116 (cit. on p. 31).

[8] J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Nonlinear Equations and Unconstrained Optimization*. Philadelphia: SIAM, 1996 (cit. on pp. 29, 30).

[9] M. Heroux et al. “An overview of the Trilinos project”. In: *ACM Trans. on Math. Software* 31.3 (2005), pp. 397–423. ISSN: 0098-3500 (cit. on pp. 29, 61).

[10] C. T. Kelley and E. W. Sachs. “A Trust Region Method for Parabolic Boundary Control Problems”. In: *SIAM J. Optimization* 9 (1999), pp. 1082–1099 (cit. on p. 31).

[11] J. Kiefer and J. Wolfowitz. “The Equivalence of Two Extremum Problems”. In: *Canadian Journal of Mathematics* 12.600 (1960), pp. 363–366. ISSN: 0008-414X. DOI: 10.4153/cjm-1960-030-4 (cit. on p. 121).

[12] D. P. Kouri et al. *Rapid Optimization Library*. 2014. URL: https://trilinos.org/packages/rol (cit. on p. 29).

[13] D.P. Kouri, J.D. Jakeman, and J.G. Huerta. “Risk-Adapted Optimal Experimental Design”. In: *SIAM/ASA Journal on Uncertainty Quantification* 10.2 (2022), pp. 687–716. DOI: 10.1137/20M1357615 (cit. on p. 121).

[14] D.P. Kouri et al. *Risk-Adaptive Experimental Design for High-Consequence Systems: LDRD Final Report.* Tech. rep. SAND2021-11380. Sandia National Laboratories, 2021. DOI: 10.2172/1820307. URL: https://www.osti.gov/biblio/1820307 (cit. on p. 121).

[15] C.-J. Lin and J. J. Moré. “Newton’s method for large bound-constrained optimization problems”. In: *SIAM J. Optim.* 9.4 (1999), pp. 1100–1127 (cit. on p. 31).

[16] R. Mayes, L. Ankers, and P. Daborn. “Predicting System Response at Unmeasured Locations”. In: *Experimental Techniques* 44.4 (2020), pp. 457–474. ISSN: 17471567. DOI: 10.1007/s40799-020-00366-9 (cit. on p. 122).

[17] Josh Netter et al. “Safe and Robust Binary Classification and Fault Detection Using Reinforcement Learning”. In: *IEEE Open Journal of Control Systems* (2025) (cit. on p. 111).

[18] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, 2006 (cit. on pp. 29, 30).

[19] J. Ranieri, A. Chebira, and M. Vetterli. “Near-Optimal Sensor Placement for Linear Inverse Problems”. In: *IEEE Transactions on Signal Processing* 62.5 (2014), pp. 1135–1146. DOI: 10.1109/TSP.2014.2299518 (cit. on p. 119).

[20] D. Ridzal, D.P. Kouri, and G. John von Winckel. *Rapid Optimization Library*. Technical report SAND2017-12025PE. Sandia National Laboratories, 2017 (cit. on p. 119).

[21] S D Team. *SD – User’s Manual*. Tech. rep. SAND2021-12052. PO Box 5800, Albuquerque, NM 87185-5800: Sandia National Laboratories, 2021 (cit. on pp. 5, 55).

[22] S D Team. *Sierra Structural Dynamics Verification*. Tech. rep. SAND2021-11330. Sandia National Laboratories, 2021 (cit. on p. 23).

[23] Sierra Thermal Fluids Team. *SIERRA Multimechanics Module: Aria User Manual - Version 5.16*. Tech. rep. SAND2023-09446O. Albuquerque, NM 87185 and Livermore, CA 94550: Sandia National Laboratories, 2023 (cit. on p. 77).

[24] Qiqi Wang, Parviz Moin, and Gianluca Iaccarino. “Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation”. In: *SIAM Journal on Scientific Computing* 31.4 (2009), pp. 2549–2567 (cit. on p. 70).

INDEX

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	T. F. Walsh	1543	0897

This page intentionally left blank.

This page intentionally left blank.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.